Fall 2023 Python Programming for Data Science

Release 0.1

Alex Vakanski

Dec 30, 2023

LECTURE 1 - SHORT HISTORY OF AI

University of Idaho - Department of Computer Science
Instructor: Alex Vakanski (vakanski@uidaho.edu)
Teaching Assistant: Longze Li (li8975@vandals.uidaho.edu)
Semester: Fall 2023 (August 21 – December 15)
Course website: https://fall-2023-python-programming-for-data-science.readthedocs.io/en/latest/
GitHub repository:
https://www.github.com/avakanski/Fall-2023-Python-Programming-for-Data-Science/blob/main/README.md

CHAPTER

ONE

COURSE SYLLABUS

View Syllabus

CHAPTER

TWO

COURSE DESCRIPTION

The course is designed to introduce students to Python tools and libraries that are commonly used by organizations for managing the various phases in the life cycle of data science projects. The content is divided into four main themes. The first theme reviews the fundamentals of Python programming. The second theme focuses on data engineering and explores Python tools for data collection, exploration, and visualization. The next theme covers model engineering and includes topics related to model design, selection, and evaluation for image processing, natural language processing, and time series analysis. The last theme introduces Data Science Operations (DSOps) and encompasses techniques for model serving, performance monitoring, diagnosis, and reproducibility of data science projects deployed in production. Throughout the course, students will gain hands-on experience with various Python libraries for data science workflow management.

CHAPTER

THREE

TEXTBOOKS

1. Joel Grus, "Data Science from Scratch: First Principles with Python", 2nd Edition, O'Reilly Media, 2019, ISBN: 9781492041139.
2. Chip Huyen, "Designing Machine Learning Systems", O'Reilly Media, 2022, ISBN: 9781098107963.

CHAPTER

FOUR

LEARNING OUTCOMES

Upon the completion of the course, the students should demonstrate the ability to:

1. Attain proficiency with commonly used Python frameworks for managing the life cycle of data science projects.
2. Develop pipelines for integrating data from multiple sources, designing predictive models, and deploying the models.
3. Apply Python tools for data collection, analysis, and visualization, such as NumPy, Pandas, Matplotlib, and Seaborn, to real-world datasets.
4. Implement machine learning algorithms for image processing, natural language processing, and time series analysis using Python-based frameworks, such as Scikit-Learn, Keras, TensorFlow, and PyTorch.
5. Understand the principles of model selection and evaluation, including hyperparameter tuning, cross-validation, and regularization.
6. Understand the primary characteristics of current Python libraries for deployment, continuous integration, and monitoring of data science projects.
7. Deploy data science projects as web applications using Flask, and to cloud servers using Microsoft's Azure platform.

CHAPTER

FIVE

PREREQUISITES

The course requires to have basic programming skills in Python. While having knowledge of data science methods would be advantageous, it is not mandatory.

CHAPTER

SIX

GRADING

Student assessment will be based on 6 homework assignments (worth 60 pts), 3 quizzes (worth 30 marks), and class participation and engagement (worth 10 marks).

CHAPTER

SEVEN

LECTURES

7.1 Lecture 1 - A Short History and Current State of Artificial Intelligence

7.2 Lecture 2 - Data Types in Python

The figure below lists the main data types in Python, and provides information about the category and mutability of the data types.

Object type	Category	Mutable?
Numbers (all)	Numeric	No
Strings (all)	Sequence	No
Lists	Sequence	Yes
Dictionaries	Mapping	Yes
Tuples	Sequence	No
Files	Extension	N/A
Sets	Set	Yes
Frozenset	Set	No
bytearray	Sequence	Yes

Figure source: Reference [1].

7.2.1 2.1 Introduction

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small-scale and large-scale projects. Python interpreters are available for many operating systems. In recent years, Python has become the primary language for Machine Learning and Data Science applications.

The Python Software Foundation (PSF) is a non-profit organization that manages and directs resources for Python development. Python 3.0 was released in 2008, and it was a significant revision of the language that is not entirely backward-compatible, and much Python 2 code does not run unmodified on Python 3. This course makes use of Python 3.

Dynamic Typing

Python uses *dynamic typing*, meaning that we can reassign different data types to variables. This makes Python very flexible in assigning data types, and it differs from other languages that are *statically typed*.

[1]:
```
# Assign the number 2 to the variable 'my_dogs'
my_dogs = 2
```

[2]:
```
# Show the variable 'my_dogs'
my_dogs
```

[2]:
```
2
```

[3]:
```
# Reassign the list to the variable 'my_dogs'
my_dogs = ['Sammy', 'Frankie']
```

[4]:
```
# Show the variable 'my_dogs'
my_dogs
```

[4]:
```
['Sammy', 'Frankie']
```

In the above example, the number `2` was first assigned to the variable `my_dogs`, and afterward the list `['Sammy', 'Frankie']` was assigned to the variable `my_dogs`.

When we create a variable name in Python, we reserve a memory location to store an object. E.g., the variable name `my_dogs` first acts as a reference to the memory location which holds the number `2`. Or, we can think of the variable as a pointer to the memory location where the value `2` is stored. Whenever we use the variable `my_dogs` in our code, Python will retrieve the value `2` from the memory location and associate it with the name `my_dogs`. By assigning the list `['Sammy', 'Frankie']` to the variable `my_dogs`, we instruct Python to associate the name `my_dogs` with a new memory location where the list `['Sammy', 'Frankie']` is stored.

Pros of Dynamic Typing

- Very easy to work with
- Faster development time

Cons of Dynamic Typing

- May result in unexpected bugs
- Requires to be aware of the type of objects

Assigning Objects to Variables

When assigning objects to variables in Python, we need to obey the following rules for the names of variable.

- Names can not start with a number
- Names can not contain spaces, use _ (underscore) instead
- Names can not contain any of these symbols `:'",<>/?|\!@#%^&*~-+`
- It's considered best practice (according to PEP8) that names are written with lowercase letters with underscores
- Avoid using Python built-in keywords like `list` and `str` in variable names
- Avoid using the single characters `l` (lowercase letter L), `O` (uppercase letter O) and `I` (uppercase letter I), since they can be confused with `1` and `0`

Variable assignment has the syntax `name = object`, where a single equal sign = is used as an **assignment operator**.

[5]:
```
# Assign the integer object 5 to the variable name 'a'
a = 5
```

[6]:
```
# Show the variable 'a'
a
```

[6]:
```
5
```

As we mentioned, dynamic typing in Python allows variables to be reassigned.

[7]:
```
a = 10
a
```

[7]:
```
10
```

Python also allows to reassign a variable with a reference to the same object.

[8]:
```
# Add 15 to the current value of 'a' and assign it to 'a'
a = a + 15
a
```

[8]:
```
25
```

Python allows using shortcuts to add, subtract, multiply, and divide numbers with re-assignment using +=, -=, *=, and /=.

For instance, `a += 10` is equivalent to `a = a + 10`

[9]:
```
# a = a + 10
a += 10
a
```

[9]:
```
35
```

[10]:
```
# a = a * 2
a *= 2
a
```

[10]:
```
70
```

Determining Variable Type with `type()`

Python offers several **built-in functions**, which can perform actions on objects. For instance, we can check the type of the object that is assigned to a variable using Python's built-in function `type()`.

[11]:
```
type(a)
```

[11]:
```
int
```

In the above example, the type of the variable `a` is *integer*.

[12]:
```
# In Python we create tuples with parantheses
a = (1,2)
```

[13]:
```
type(a)
```

[13]:
```
tuple
```

It is also important to note that the double equal == operator in Python is used to test the equality of two expressions, whereas the single equal = operator is used to assign objects to variables.

Also, testing for inequality is performed with the not equal `!=` operator.

The operators greater than >, less than <, greater than or equal >=, less than or equal <= perform as would generally be expected.

7.2.2 2.2 Numbers

Integers are whole numbers, and can be positive or negative. For example: 2 and -2.

Floating point numbers in Python have a decimal point, or use an exponential (E) to define the number. For example 2.0 and -2.163 are examples of floating point numbers. `4E2` (4 times 10 to the power of 2 = 400) and `1E-3` (3 times 10 to the power of -3 = 0.001) are also examples of floating point numbers in Python.

Other types of number objects that are less frequently used include:

- **Complex numbers**, have real and imaginary parts, e.g., 3+4j
- **Decimal numbers**, have control over the precision and rounding of numbers, e.g., `Decimal('0.1')` (see examples in the next section)
- **Fractions**, are rational numbers with numerator and denominator, e.g., `Fraction(1,3)` = 1/3.

Basic Arithmetic Operations

[14]:
```
# Addition
2+1
```

[14]:
```
3
```

[15]:
```
# Subtraction
2-1
```

[15]:
```
1
```

[16]:
```
# Multiplication
2*2
```

[16]:
```
4
```

[17]:
```
# Division
3/2
```

[17]:
```
1.5
```

[18]:
```
# Floor Division
7//4
```

[18]:
```
1
```

The floor division operator `//` (two forward slashes) truncates the decimal number without rounding, and returns an integer result.

If we just want the remainder after division, we use the `%` modulo operator.

[19]:
```
# Modulo (remainder)
7%4
```

[19]:
```
3
```

[20]:
```
# Power (exponentiation)
2**3
```

[20]:
```
8
```

[21]:
```
# Can also do roots (e.g., square root is **0.5)
4**0.5
```

[21]:
```
2.0
```

[22]:
```
# Order of operations followed in Python
# Precedence: Parenthesis, Exponentiation, Division, Multiplication, Addition,␣
↪Subtraction
# E.g., multiplication has precedence over addition
2 + 10 * 10 + 3
```

[22]:
```
105
```

[23]:
```
# Can use parentheses to specify orders
(2+10) * (10+3)
```

[23]:
```
156
```

Note that floating-point numbers are implemented in computer hardware as binary fractions (fractions of 0 and 1). As a result, many decimal fractions cannot be accurately represented as binary fractions. For example, the decimal number 0.1 results in an infinitely long binary fraction of 0.0001100110011001100110011…. Since our computer can only store a finite number of decimal places, this will only approximate the above binary fraction, but the approximation will not be equal to 0.1. Hence, such approximations of decimal numbers is the limitation of our computer hardware and not an error in Python.

[24]:
```
# Note that display issue in Python, due to using binary fractions to represent float␣
↪numbers
f3 = 0.1 + 0.1 + 0.1
f3
```

[24]:
```
0.30000000000000004
```

[25]:
```
# One solution to that is to use decimal numbers, since they have rounding mechanisms to␣
↪obtain exact representations
from decimal import Decimal
f4 = Decimal('0.1') + Decimal('0.1') + Decimal('0.1')
f4
```

[25]:
```
Decimal('0.3')
```

If number types are mixed, Python will do the conversion.

[26]:
```
# Mix int and float; Python will convert int to float first
a = 1 + 2.5
print(a)
type(a)
```

```
3.5
```

[26]:
```
float
```

[27]:
```
# Convert between different types
a = 2
```

(continues on next page)

(continued from previous page)

```
b = float(a)
print(b)
type(b)
```

```
2.0
```

[27]:
```
float
```

[28]:
```
c = int(b)
print(c)
type(c)
```

```
2
```

[28]:
```
int
```

We can also use logic comparisons with numbers using <, >, >=, <=.

[29]:
```
# Logic comparison
5<3
```

[29]:
```
False
```

Built-in Mathematical Functions

Examples of *built-in mathematical functions* include: `pow`, `abs`, `round`, and others. These functions are built into the Python interpreter. We do not need to import any packages. Check the list of all built-in functions in Python: https://docs.python.org/3.10/library/functions.html

[30]:
```
# Power (exponentiation)
pow(2,4)
```

[30]:
```
16
```

[31]:
```
round(3.006)
```

[31]:
```
3
```

[32]:
```
# Absolute value
abs(-3.4)
```

[32]:
```
3.4
```

[33]:
```
# Check documentation for help about built-in functions
help(pow)
```

```
Help on built-in function pow in module builtins:

pow(base, exp, mod=None)
    Equivalent to base**exp with 2 arguments or base**exp % mod with 3 arguments

    Some types, such as ints, are able to use a more efficient algorithm when
    invoked using the three argument form.
```

Python Modules for Numerical Operations

The number of built-in mathematical functions is limited, and we can also import Python modules, such as `math` and `random` to perform mathematical operations. https://docs.python.org/3.10/library/math.html#module-math

[34]:
```
import math

math.floor(3.006)
```

[34]:
```
3
```

[35]:
```
import random

# Return a random floating-point number in the range 0-1
r = random.random()
print(r)
```

```
0.7360686591339091
```

7.2.3 2.3 Strings

A **string** is an immutable sequence containing letters, words, and other characters.

Strings are used in Python to record text information, such as names. Strings in Python are *sequences*, which means that Python keeps track of every element in the string, and we can use indexing to get particular elements in the sequence.

Creating a String

To create a string in Python we can use either single quotes or double quotes.

[36]:
```
# Single word
'hello'
```

[36]:
```
'hello'
```

[37]:
```
# Entire phrase
'This is also a string'
```

[37]:
```
'This is also a string'
```

[38]:
```
# We can also use double quotes
"String built with double quotes"
```

[38]:
```
'String built with double quotes'
```

Note that the code below shows an error, because the single quote in `I'm` broke the continuation of the single quotes in the string.

[39]:
```
# Be careful with quotes!
' I'm using single quotes, but this will create an error'
```

```
  File "C:\Users\Alex\AppData\Local\Temp\ipykernel_9980\2053197537.py", line 2
    ' I'm using single quotes, but this will create an error'
       ^
```

(continues on next page)

(continued from previous page)

```
SyntaxError: invalid syntax
```

You can use combinations of double and single quotes to get the complete statement.

[40]:
```
"Now I'm ready to use the single quotes inside a string!"
```

[40]:
```
"Now I'm ready to use the single quotes inside a string!"
```

Printing a String

In Jupyter notebooks, writing a string in a cell will automatically output the string, however the correct way to display strings is by using a **print** function.

[41]:
```
# We can simply declare a string
'Hello World'
```

[41]:
```
'Hello World'
```

[42]:
```
# Note that we can't output multiple strings this way; only the last string is displayed
'Hello World 1'
'Hello World 2'
```

[42]:
```
'Hello World 2'
```

We can use a `print` statement to display a string, or multiple strings in a cell.

[43]:
```
print('Hello World 1')
print('Hello World 2')
print('Use \n to print a new line') # \n prints a new line
print('\n')
print('See what I mean?')
```

```
Hello World 1
Hello World 2
Use
 to print a new line


See what I mean?
```

We can also use the built-in function **len()** to check the length of a string. It counts all of the characters in the string, including spaces and punctuation marks.

[44]:
```
len('Hello World')
```

[44]:
```
11
```

String Indexing and Slicing

Since strings are sequences, Python can use indexes to call parts of the sequence.

Indexing starts at 0 for Python.

[45]:
```
# Assign a string to the variable 's'
s = 'Hello World'
```

[46]:
```
# Show 's'
s
```

[46]:
```
'Hello World'
```

[47]:
```
# Print the string
print(s)
```

```
Hello World
```

[48]:
```
# Check the type of 's'
type(s)
```

[48]:
```
str
```

[49]:
```
# Show first element
s[0]
```

[49]:
```
'H'
```

Use the slicing operator `:` to perform **slicing** which returns the elements up to a designated index in the sequence.

[50]:
```
# Grab everything past the first element all the way to the end of s
s[1:]
```

[50]:
```
'ello World'
```

[51]:
```
# Note that there is no change to the original s
s
```

[51]:
```
'Hello World'
```

[52]:
```
# Grab everything UP TO the 3rd index
s[:3]
```

[52]:
```
'Hel'
```

The above slicing includes indexes 0, 1, and 2, and it doesn't include the 3rd index. In Python, slicing is performed as **up to, but not including**.

[53]:
```
# Return everything
s[:]
```

[53]:
```
'Hello World'
```

We can also use negative indexing to go backwards.

[54]:
```
# Last letter (one index behind 0 so it loops back around)
s[-1]
```

[54]:
```
'd'
```

[55]:
```
# Grab everything but the last letter
s[:-1]
```

[55]:
```
'Hello Worl'
```

We can also use indexing and slicing notation to grab elements of a sequence by a specified step size (the default is 1). For instance, we can use two colons in a row `::` and then a number specifying the frequency to grab elements.

[56]:
```
# Grab everything, but go in steps size of 1
s[::1]
```

[56]:
```
'Hello World'
```

[57]:
```
# Grab everything, but go in step sizes of 2
s[::2]
```

[57]:
```
'HloWrd'
```

[58]:
```
# We can use step size of -1 to print a string backwards
s[::-1]
```

[58]:
```
'dlroW olleH'
```

String Properties

Strings are *immutable* objects. It means that once a string is created, the elements within it **can not** be changed or replaced.

[59]:
```
s
```

[59]:
```
'Hello World'
```

[60]:
```
# Let's try to change the first letter to 'x'
s[0] = 'x'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9980\2138848189.py in <module>
      1 # Let's try to change the first letter to 'x'
----> 2 s[0] = 'x'

TypeError: 'str' object does not support item assignment
```

Other properties of strings include concatenation, i.e., we can concatenate strings.

[61]:
```
s
```

[61]:
```
'Hello World'
```

[62]:
```
# Concatenate strings
s + ' concatenate me!'
```

[62]:
```
'Hello World concatenate me!'
```

[63]:
```
# We can reassign s completely
s = s + ' concatenate me!'
```

[64]:
```
# Note that now s points to the entire sequence
print(s)
```

```
Hello World concatenate me!
```

We can also use the multiplication symbol * to create a repetition of a string.

[65]:
```
letter = 'z'
letter*10
```

[65]:
```
'zzzzzzzzzz'
```

Built-in Methods for Strings

Objects in Python can also have **built-in methods**. Methods are called with a period followed by the method name, as in:

`object.method(parameters)`

In the above line, `parameters` are extra arguments we can pass into the method.

Here are some examples of built-in methods in strings.

[66]:
```
s
```

[66]:
```
'Hello World concatenate me!'
```

[67]:
```
# Upper case the string
s.upper()
```

[67]:
```
'HELLO WORLD CONCATENATE ME!'
```

[68]:
```
# Lower case
s.lower()
```

[68]:
```
'hello world concatenate me!'
```

[69]:
```
# Split a string by blank spaces (this is the default)
s.split()
```

[69]:
```
['Hello', 'World', 'concatenate', 'me!']
```

[70]:
```
# Split by a specific element (doesn't include the element that was split on)
s.split('W')
```

[70]:
```
['Hello ', 'orld concatenate me!']
```

[71]:
```
# Check all built-in methods for the string s
dir(s)
```

[71]:
```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
 'isalnum',
 'isalpha',
 'isascii',
 'isdecimal',
 'isdigit',
 'isidentifier',
```

(continues on next page)

(continued from previous page)

```
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'removeprefix',
 'removesuffix',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

7.2.4 2.4 Lists

A **list** is a mutable ordered sequence of elements, written as a series of items within square brackets.

Lists are the most general example of a *sequence* in Python. Unlike strings, they are mutable, meaning that the elements inside a list can be changed.

Lists are constructed with square brackets `[]` and commas separating every element in the list.

[72]:
```
# Assign a list to the variable named 'my_list'
my_list = [1, 2, 3]
```

Lists can hold different object types. For example, the following list contains strings, integers, and float numbers.

[73]:
```
my_list = ['A string', 23, 100.232, 'o']
my_list
```

[73]:
```
['A string', 23, 100.232, 'o']
```

Just like with strings, the built-in function `len()` returns the number of items in the sequence of the list.

[74]:
```
len(my_list)
```

[74]:
```
4
```

We can think of lists as arrays of references (pointers) to a series of objects with allocated memory.

List Indexing and Slicing

Indexing and slicing work just like in strings.

[75]:
```
my_list = ['one', 'two', 'three', 4, 5]

# Grab element at index 0
my_list[0]
```

[75]:
```
'one'
```

[76]:
```
# Grab everything UP TO index 3
my_list[:3]
```

[76]:
```
['one', 'two', 'three']
```

We can also use + to concatenate lists, just like we did for strings.

[77]:
```
my_list + ['new item']
```

[77]:
```
['one', 'two', 'three', 4, 5, 'new item']
```

Note that the above operation doesn't actually change the original list.

[78]:
```
my_list
```

[78]:
```
['one', 'two', 'three', 4, 5]
```

To make the change permanent, we need to reassign the list.

[79]:
```
# Reassign
my_list = my_list + ['add new item permanently']
```

[80]:
```
my_list
```

[80]:
```
['one', 'two', 'three', 4, 5, 'add new item permanently']
```

We can also use the operator * for a duplication method similar to strings.

[81]:
```
# Make the list double
my_list * 2
```

[81]:
```
['one',
 'two',
 'three',
 4,
 5,
 'add new item permanently',
 'one',
 'two',
```

(continues on next page)

(continued from previous page)

```
 'three',
 4,
 5,
 'add new item permanently']
```

[82]:
```
# Again, doubling is not permanent
my_list
```

[82]:
```
['one', 'two', 'three', 4, 5, 'add new item permanently']
```

Lists indexing will return an error if there is no element at that index. For example:

[83]:
```
my_list[100]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9980\3678114954.py in <module>
----> 1 my_list[100]

IndexError: list index out of range
```

Built-in Methods for Lists

There are parallels between arrays in other programming languages and lists in Python. Lists in Python tend to be more flexible than arrays in other languages for two reasons: they have no fixed size (meaning we don't have to specify the size of a list when we create one), and they have no fixed type constraint (like we've seen above we can mix different types of objects in a list).

Explained next are several built-in methods for lists.

[84]:
```
# Create a new list
list1 = [1, 2, 3]
```

Use the **append** method to permanently add an item to the end of a list:

[85]:
```
# Append
list1.append('append me!')
```

[86]:
```
# Show
list1
```

[86]:
```
[1, 2, 3, 'append me!']
```

Use **pop** to extract ("pop off") an item from the list. By default, pop takes off the last index (i.e., with index -1), but we can also specify which index to pop off.

[87]:
```
# Pop off the 0 indexed item
list1.pop(0)
```

[87]:
```
1
```

[88]:
```
# Show
list1
```

[88]:
```
[2, 3, 'append me!']
```

[89]:
```
# Assign the popped element; remember that the default popped index is -1
popped_item = list1.pop()
```

[90]:
```
popped_item
```

[90]:
```
'append me!'
```

[91]:
```
# Show remaining list
list1
```

[91]:
```
[2, 3]
```

We can insert and remove elements from a list.

[92]:
```
c = ['a', 'b', 'c']

# Insert at index 0
c.insert(0, 'a0')
print(c)
```

```
['a0', 'a', 'b', 'c']
```

[93]:
```
# Remove
c.remove('b')
print(c)
```

```
['a0', 'a', 'c']
```

[94]:
```
# Remove at index 0
del c[0]
print(c)
```

```
['a', 'c']
```

We can use the **sort** and **reverse** methods with lists.

[95]:
```
new_list = ['a','e','x','b','c']
```

[96]:
```
# Show
new_list
```

[96]:
```
['a', 'e', 'x', 'b', 'c']
```

[97]:
```
# Use reverse to reverse order (this is permanent!)
new_list.reverse()
```

[98]:
```
new_list
```

[98]:
```
['c', 'b', 'x', 'e', 'a']
```

[99]:
```
# Use sort to sort the list (in alphabetical order)
new_list.sort()
```

[100]:
```
new_list
```

[100]:
```
['a', 'b', 'c', 'e', 'x']
```

[101]:
```
# For list of numbers, sorting is in ascending order
list_of_numbers = [2, 4, 3, 7, 1]
list_of_numbers.sort()
list_of_numbers
```

[101]:
```
[1, 2, 3, 4, 7]
```

Two lists can be combined into a single list by the **zip** function.

[102]:
```
a = [1, 2, 3, 4, 5]
b = [5, 4, 3, 2, 1]

print(zip(a,b))
```

```
<zip object at 0x0000028AC666CFC0>
```

To see the results of the **zip** function, in the next cell we convert the returned zip object into a list. Note that the **zip** function returns a list of tuples. Each tuple represents a pair of items that the function zipped together. The order in the two lists was maintained.

[103]:
```
print(list(zip(a,b)))
```

```
[(1, 5), (2, 4), (3, 3), (4, 2), (5, 1)]
```

Often the **zip** command is used inside of a for-loop. The following code shows how a for-loop can assign a variable to each collection that the program is iterating.

[104]:
```
a = [1, 2, 3, 4, 5]
b = [5, 4, 3, 2, 1]

for x,y in zip(a,b):
    print(f'{x} - {y}')
```

```
1 - 5
2 - 4
3 - 3
4 - 2
5 - 1
```

Nesting Lists

Python data structures support **nesting**, that is, we can have data structures within data structures. For example, a list inside a list is shown next.

[105]:
```
# Let's make three lists
lst_1 = [1, 2, 3]
lst_2 = [4, 5, 6]
lst_3 = [7, 8, 9]

# Make a list of lists to form a matrix
matrix = [lst_1, lst_2, lst_3]
```

[106]:
```
# Show
matrix
```

[106]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

We can again use indexing to grab elements, but now there are two levels for the index: the items in the matrix object, and the items inside that list.

[107]:
```
# Grab the first item in the matrix object
matrix[0]
```

[107]:
```
[1, 2, 3]
```

[108]:
```
# Grab the first item of the first item in the matrix object
matrix[0][0]
```

[108]:
```
1
```

List Comprehensions

Python has an advanced feature called **list comprehensions**. This allows for quick construction of lists.

The basic syntax of list comprehensions is:

`newlist = [expression for item in iterable]`

It returns a new list by applying the expression to each item in the iterable object.

[109]:
```
# Apply x+10 to each element in the list
list2 = [1, 2, 3]
new_list = [x+10 for x in list2]
new_list
```

[109]:
```
[11, 12, 13]
```

We can also add logic conditions to the list comprehensions as in the next example.

[110]:
```
# Even numbers in the range 0-10
even_numbers = [x for x in range(10) if x%2 == 0]
even_numbers
```

[110]:
```
[0, 2, 4, 6, 8]
```

[111]:
```
# Squared elements in the even numbers list
even_squares = [x*x for x in even_numbers]
even_squares
```

[111]:
```
[0, 4, 16, 36, 64]
```

7.2.5 2.5 Dictionaries

A **dictionary** is an unordered and mutable Python container that stores mappings of unique keys to values.

We've been learning about *sequences* in Python so far, and now we're going to learn about *mappings* in Python. If you're familiar with other programming languages, you can think of dictionaries as hash tables.

Mappings are a collection of objects that are stored by a unique *key*, unlike a sequence that stores objects by their relative position (index). This is an important distinction, since mappings won't retain the order of the elements because each object is defined by a key.

Therefore, a Python dictionary consists of a collection of **keys** and associated **values**. A colon `:` separates each key from its value, and the keys and values are provided inside curly braces `{}`. The keys must be unique, and can appear only once in a dictionary. Also, the keys must be immutable objects, such as strings, integers, floats, or tuples. The associated values can be almost any Python object, and there are no restrictions.

Dictionaries are mutable, therefore the elements can be changed, added, and removed.

Constructing a Dictionary

[112]:
```
# Make a dictionary with {} and : to signify a key and a value
my_dict = {'key1':'value1','key2':'value2'}
```

[113]:
```
my_dict
```

[113]:
```
{'key1': 'value1', 'key2': 'value2'}
```

[114]:
```
# Call values by their key
my_dict['key2']
```

[114]:
```
'value2'
```

Dictionaries are very flexible and they can hold various data types.

[115]:
```
# The values can be any object type
my_dict = {1:101, 2:102, 3:103}

# Call an item
my_dict[2]
```

[115]:
```
102
```

[116]:
```
my_dict = {'key1':123, 'key2':[12,23,33], 'key3':['item0','item1','item2']}

# Let's call items from the dictionary
my_dict['key3']
```

[116]:
```
['item0', 'item1', 'item2']
```

[117]:
```
# We can call an index on that value
my_dict['key3'][0]
```

[117]:
```
'item0'
```

[118]:
```
# Can even call methods on that value
my_dict['key3'][0].upper()
```

[118]:
```
'ITEM0'
```

We can change the values in a dictionary, as in the following cell.

[119]:
```
# Subtract 123 from the value
print(my_dict['key1'])
my_dict['key1'] = my_dict['key1'] - 100
```

```
123
```

[120]:
```
#Check
my_dict['key1']
```

[120]:
```
23
```

The keys in Python dictionaries are immutable, and we cannot change them. However, we can remove a key using `pop()` or `del` as in lists, and afterward add a new key with its associated value.

We can also create new keys and values by assignment. For instance, if we start with an empty dictionary, we can continually add key-value pairs to it.

[121]:
```
# Create a new dictionary
d = {}
```

[122]:
```
# Create a new key through assignment
d['animal'] = 'Dog'
```

[123]:
```
# Can do this with any object
d['answer'] = 42
```

[124]:
```
# Show
d
```

[124]:
```
{'animal': 'Dog', 'answer': 42}
```

Nesting with Dictionaries

Python has flexibility of nesting objects and calling methods on them. Let's see a dictionary nested inside a dictionary.

[125]:
```
# Dictionary nested inside a dictionary nested inside a dictionary
d = {'key1':{'nestkey':{'subnestkey':32}}}
```

[126]:
```
# Keep calling the keys
d['key1']['nestkey']['subnestkey']
```

[126]:
```
32
```

Dictionary Built-In Methods

There are several built-in methods we can call on a dictionary.

[127]:
```
# Create a dictionary
d = {'key1':1,'key2':2,'key3':3}
```

[128]:
```
# Method to return a list of all keys
d.keys()
```

[128]:
```
dict_keys(['key1', 'key2', 'key3'])
```

[129]:
```
# Method to return all values
d.values()
```

[129]:
```
dict_values([1, 2, 3])
```

7.2.6 2.6 Tuples

> A **tuple** is a collection of objects which is ordered and immutable, and it is commonly written as a series of items in parentheses.

In Python, tuples are very similar to lists, with the main difference being that tuples are *immutable* sequences, unlike lists that are *mutable* sequences. Tuples are created similarly to lists, but with parantheses `()` instead of squared brackets `[]`.

The basic characteristics of tuples include:

- They are ordered collections of objects: like lists and strings, tuples are positionally ordered collections of objects (i.e., they are sequences) that maintain a left-to-right order among their elements.
- Are accessed by offset: like strings and lists, items in a tuple are accessed by positional offset (not by key); therefore, they support indexing and slicing.
- Tuples are immutable sequences: like strings and lists, tuples are sequences. However, unlike lists that are *mutable* sequences, tuples are *immutable* sequences (meaning they can not be changed in place).
- Are fixed-length, heterogeneous, and arbitrarily nestable: because tuples are immutable, their size cannot be changed (without making a new copy). Tuples can hold any type of object, including other compound objects (e.g., lists, dictionaries, other tuples), and hence, they support arbitrary nesting.
- Tuples are arrays of object references: like lists, tuples are best thought of as arrays of references (pointers) to other objects with allocated memory.

Constructing Tuples

Tuples are constructed by using parentheses `()` with the items separated by commas.

[130]:
```
# Creating a tuple
t = (1, 2, 3)
t
```

[130]:
```
(1, 2, 3)
```

[131]:
```
# Check the length of the tuple using len(), just like a list
len(t)
```

[131]:
```
3
```

[132]:
```
# We can also mix object types: e.g., strings, integer numbers, floating-point numbers
t = ('one', 2, 490.2)

# Show
t
```

[132]:
```
('one', 2, 490.2)
```

[133]:
```
# Tuples, lists, or dictionaries can be nested into other tuples
w = ('one', 'two', (4, 5), 6, ['r', 100])
w
```

[133]:
```
('one', 'two', (4, 5), 6, ['r', 100])
```

[134]:
```
# An empty tuple
u = ()
u
```

[134]:
```
()
```

[135]:
```
# A 1-item tuple
v = ('thing', )
v
```

[135]:
```
('thing',)
```

Note that for a single-item tuple we need to place a comma after the item, that is, we use `(item,)` and not `(item)`, since parentheses can also be used to enclose expressions like `(1 + 2) * 3` = 9.

[136]:
```
# Note that the output of this cell is not a tuple
# Since the displayed output of the cell is not in parentheses, it is an integer number, not a tuple
a = (3)
a
```

[136]:
```
3
```

We can also use the built-in function `type()` to check the type of the variable `a`.

[137]:
```
# The type of the variable a is integer number
type(a)
```

[137]:
```
int
```

[138]:
```
# This is a tuple
b = (3,)
b
```

[138]:
```
(3,)
```

[139]:
```
# The type of the variable b is tuple
type(b)
```

[139]:
```
tuple
```

[140]:
```
# Not a tuple
(1 + 4) * 3
```

[140]:
```
15
```

[141]:
```
# This is a tuple: note that (1+4,) is the same as (5,), and when multiplied by 3, the␣
↪tuple is repeated 3 times
(1 + 4,) * 3
```

[141]:
```
(5, 5, 5)
```

The parentheses () can be omitted in the syntax, and tuples in Python can be created just by listing items separated with commas. Although the parentheses are mostly optional with tuples, there are a few cases when using parentheses is required, e.g., within a function call, or when nested in a larger expression. For beginners, it is recommended to always use parentheses, in order to avoid the above exceptions, and because they improve the code readability.

[142]:
```
t = 'one', 2, 490.2
t
```

[142]:
```
('one', 2, 490.2)
```

[143]:
```
# A tuple with one item can be created just by adding a comma after the item without␣
↪using parentheses
'hello',
```

[143]:
```
('hello',)
```

Tuple Indexing and Slicing

Since tuples are positionally ordered collections of objects like strings and lists, indexing and slicing work for tuples.

[144]:
```
t
```

[144]:
```
('one', 2, 490.2)
```

[145]:
```
# Use indexing just like in lists and strings
t[0]
```

[145]:
```
'one'
```

[146]:
```
t[1]
```

[146]:
```
2
```

[147]:
```
t[-1]
```

[147]:
```
490.2
```

[148]:
```
# Slicing
t[0:2]
```

[148]:
```
('one', 2)
```

Other sequencing operations, such as concatenation and repetition, are also supported for tuples, in a similar way as for lists and strings.

[149]:
```
# Concatenation
(1, 'book') + ('notes', 4)
```

[149]:
```
(1, 'book', 'notes', 4)
```

[150]:
```
# Repetition
(1, 'thing') * 4
```

[150]:
```
(1, 'thing', 1, 'thing', 1, 'thing', 1, 'thing')
```

Because tuples are sequences, we can also use `for` loop iterations and list comprehensions to print the elements of tuples.

[151]:
```
# Consider the following tuple
x = ('b', 'u', 'i', 'l', 'd', 'i', 'n', 'g')
x
```

[151]:
```
('b', 'u', 'i', 'l', 'd', 'i', 'n', 'g')
```

[152]:
```
# We can use a `for` loop iteration to print each of the items of the tuple on a separate␣
↪line
for i in x:
    print(i)
```

```
b
u
i
l
d
i
n
g
```

[153]:
```
# A list comprehension can also be used to print each of the items of the tuple x on a␣
↪separate line
l = [print(i) for i in x]
```

```
b
u
i
l
d
i
n
g
```

Built-in Methods for Tuples

There are built-in methods for tuples in Python, but not as many as for lists. Tuples do not have methods such as `append()`, `remove()`, `extend()`, `insert()`, and `pop()` due to their immutable nature.

[154]:
```
# Show
t
```

[154]:
```
('one', 2, 490.2)
```

[155]:
```
# Use .index to enter an item and return the index
t.index('one')
```

[155]:
```
0
```

[156]:
```
# Use .count to count the number of times a value appears
t.count('one')
```

[156]:
```
1
```

[157]:
```
# Count the number of times 2 appears in the tuple
u = (1, 2, 3, 2, 1, 2)
u.count(2)
```

[157]:
```
3
```

Tuple Immutability

To emphasize one more time that tuples are immutable, check the following examples.

[158]:
```
# If we try to change the first element, we will get an error message
t[0] = 'four'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9980\3799902626.py in <module>
      1 # If we try to change the first element, we will get an error message
----> 2 t[0] = 'four'

TypeError: 'tuple' object does not support item assignment
```

Because of their immutability, tuples can't grow. Once a tuple is created, we can not add to it.

[159]:
```
# We will get an error message
t.append('nope')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9980\402286559.py in <module>
      1 # We will get an error message
----> 2 t.append('nope')

AttributeError: 'tuple' object has no attribute 'append'
```

We can, however, make a new tuple based on a current tuple.

[160]:
```
t = (t[0], 7, t[2])
t
```

[160]:
```
('one', 7, 490.2)
```

Conversion to Lists

Conversion to lists and back to tuples is straightforward.

[161]:
```
type(list)
```

[161]:
```
type
```

[162]:
```
# Tuple to list
l = list(t)
l
```

[162]:
```
['one', 7, 490.2]
```

[163]:
```
# List to tuple
l2 = ['aa', 'bb', 5, 'cc']
t2 = tuple(l2)
t2
```

[163]:
```
('aa', 'bb', 5, 'cc')
```

Tuple Unpacking

Tuple **unpacking** means pairing objects on the right side of the assignment operator = with targets on the left side by position, and assigning them from left to right.

[164]:
```
# Unpacking the tuple into the individual items
y = ('GOOG', 120, 490.2)
order, shares, price = y
print(order)
print(shares)
print(price)
```

```
GOOG
120
490.2
```

[165]:
```
print('Cost:', shares * price)
```

```
Cost: 58824.0
```

[166]:
```
# Unpacking the tuple: two names are entered for a tuple with 3 items, resulting in an␣
↪error
order, shares = y
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9980\526350587.py in <module>
```

(continues on next page)

(continued from previous page)

```
      1 # Unpacking the tuple: two names are entered for a tuple with 3 items, resulting␣
↪in an error
----> 2 order, shares = y

ValueError: too many values to unpack (expected 2)
```

Named Tuples

Named tuples are an extended type of tuples that allow items to be accessed by both position and attribute name, similar to dictionaries. They are created by using the `namedtuple` function from the `collections` module.

[167]:
```
# Import and create a named tuple
from collections import namedtuple

Rec = namedtuple('Record', ['name', 'age', 'jobs'])

# Assign a named-tuple record
bob = Rec(name='Bob', age=40.5, jobs=['dev', 'mgr'])
alice = Rec(name='Alice', age=36, jobs=['mgr'])
```

[168]:
```
# Access by position
bob[0]
```

[168]:
```
'Bob'
```

[169]:
```
bob[1]
```

[169]:
```
40.5
```

[170]:
```
# Access by attribute
bob.name, bob.jobs
```

[170]:
```
('Bob', ['dev', 'mgr'])
```

A named tuple can be converted to a dictionary, which allows key-based access to the items.

[171]:
```
D = bob._asdict()
D
```

[171]:
```
{'name': 'Bob', 'age': 40.5, 'jobs': ['dev', 'mgr']}
```

[172]:
```
# Access by key
D['name']
```

[172]:
```
'Bob'
```

When to Use Tuples

Although tuples are very similar to lists, tuples are not used as often as lists in programming. However, tuples are used when immutability is necessary; for instance, if in your program you are using an object and need to make sure it does not get changed, then a tuple provides convenient integrity.

7.2.7 2.7 Sets

> A **set** is a collection of *unique* objects which is unordered and mutable, and are constructed by using the set() function.

Sets support operations corresponding to mathematical set theory, such as intersection, union, etc. By definition, an item appears only once in a set, no matter how many times it is added.

Because sets are collections of objects, they share some behavior with lists and dictionaries. For example, sets are iterable, can grow and shrink on demand, and may contain a variety of object types.

However, since sets are unordered and do not map keys to values, they are neither a sequence nor mapping type.

Sets have a variety of applications, especially in numeric and database-focused work.

To create a set object, pass in a sequence or another iterable object to the built-in set function.

[173]:
```
x = set('abcde')
x
```

[173]:
```
{'a', 'b', 'c', 'd', 'e'}
```

The sets are displayed with curly brackets. This is similar to a dictionary, but sets do not have keys and values (or, they can be considered dictionaries with only keys and without any values).

Set Expressions

[174]:
```
x = set('abcde')
y = set('bdxyz')
```

[175]:
```
# Union
x | y
```

[175]:
```
{'a', 'b', 'c', 'd', 'e', 'x', 'y', 'z'}
```

[176]:
```
# Intersection
x & y
```

[176]:
```
{'b', 'd'}
```

[177]:
```
# Difference
x - y
```

[177]:
```
{'a', 'c', 'e'}
```

[178]:
```
# Symmetric difference (XOR) - elements in either x or y, but not both in x and y
x ^ y
```

[178]:
```
{'a', 'c', 'e', 'x', 'y', 'z'}
```

[179]:
```
# Superset, subset
x > y, x < y
```

[179]:
```
(False, False)
```

[180]:
```
# Membership of a set
'e' in x
```

[180]:
```
True
```

Sets can also be created by adding elements to an existing set object.

[181]:
```
# Create a set
z = set()
```

[182]:
```
# Add to set with the add() method
z.add(1)
z
```

[182]:
```
{1}
```

[183]:
```
# Add a different element
z.add(2)
z
```

[183]:
```
{1, 2}
```

[184]:
```
# Try to add the same element
z.add(1)
z
```

[184]:
```
{1, 2}
```

We cannot add another 1, because a set has only unique elements.

For instance, we can cast a list with multiple repeat elements into a set to get the unique elements of the list.

[185]:
```
# Create a list with repeats
list1 = [1, 1, 2, 2, 3, 4, 5, 6, 1, 1]
# Cast as set to get unique values
set(list1)
```

[185]:
```
{1, 2, 3, 4, 5, 6}
```

Built-in Methods for Sets

Similar to the set expressions shown above, there are built-in set methods for union, intersection, and other related operations.

[186]:
```
x = set('abcde')
y = set('bdxyz')
# Same as x & y
z = x.intersection(y)
z
```

[186]:
```
{'b', 'd'}
```

[187]:
```
# Delete one item
z.remove('b')
z
```

[187]:
```
{'d'}
```

Also, we can use for-loops with the elements of sets.

[188]:
```
for item in set('abc'):
    print(item * 3)
```

```
aaa
ccc
bbb
```

7.2.8 2.8 Other Data Types

Booleans

Python also has a Boolean data type with predefined built-in names `True` and `False`, that are basically just the integers 1 and 0.

[189]:
```
# Assign the Boolean True object to 'a'
a = True
# Show
a
```

[189]:
```
True
```

The data type for `True` and `False` is `bool`.

[190]:
```
type(a)
```

[190]:
```
bool
```

We can also use comparison operators to create Booleans.

[191]:
```
# Output is boolean
1 > 2
```

[191]:
```
False
```

[192]:
```
# Is True the same as 1
True == 1
```

[192]:
```
True
```

In Python each object is either True or False, as follows:

- Numbers are false if zero, and true otherwise.
- Other objects are false if empty, and true otherwise.

[193]:
```
bool(2)
```
[193]:
```
True
```

[194]:
```
bool(0)
```
[194]:
```
False
```

[195]:
```
bool('book')
```
[195]:
```
True
```

[196]:
```
bool('')
```
[196]:
```
False
```

[197]:
```
bool([1, 2])
```
[197]:
```
True
```

The None Object

We can use **None** as a placeholder for an object that we don't want to reassign yet.

[198]:
```
# None placeholder
b = None
```

[199]:
```
# Show
print(b)
```
```
None
```

For instance, to initialize a list whose size is not known yet, we can use `None` to preset the initial size and allow for future index assignment.

Note also that in Python, the fixed values or the raw data that are assigned to variables or constants are called **literals**. Examples of Python literals include: numeric literals (e.g., the integer number `5`, or float number `2.1`), string literals (e.g., any string like `'hi'` or `"hello"`), Boolean literals (`True` and `False`), special literal `None`, list literals (e.g., the list `[1, 2, 3]`, etc. Therefore, a literal is a sequence of characters or a single character that represents a fixed value in source code.

7.2.9 2.9 String Formatting

String formatting allows injecting items into a string, rather than trying to chain items together using commas or string concatenation. As a quick comparison, consider:

```
player = 'Thomas'
points = 33

# concatenation
'Last night, '+player+' scored '+str(points)+' points.'

# string formatting
```

(continues on next page)

(continued from previous page)

```
f'Last night, {player} scored {points} points.'

# The output of both concatenation and string formatting is the same:
'Last night Thomas scored 33 points.'
```

There are three ways to perform string formatting.

- The oldest method involves placeholders using the modulo `%` character.
- An improved technique uses the `.format()` string method.
- The newest method, introduced with Python 3.6, uses formatted string literals, called `f-strings`.

These three methods are described next.

Formatting with Placeholders

We can use `%s` to inject strings into print statements. The modulo `%` is referred to as a **string formatting operator**.

[200]:
```
print("I'm going to inject %s here." %'something')
```

```
I'm going to inject something here.
```

We can pass multiple items by placing them inside a tuple after the `%` operator.

[201]:
```
print("I'm going to inject %s text here, and %s text here." %('some','more'))
```

```
I'm going to inject some text here, and more text here.
```

We can also pass variable names.

[202]:
```
x, y = 'some', 'more'
print("I'm going to inject %s text here, and %s text here."%(x,y))
```

```
I'm going to inject some text here, and more text here.
```

Note that there are two notations `%s` and `%r` that convert any Python object to a string using two separate methods: `str()` and `repr()`. Here, `%r` and `repr()` deliver the *string representation* of the object, including quotation marks and any escape characters.

[203]:
```
print('He said his name was %s.' %'Fred')
print('He said his name was %r.' %'Fred')
# Note that in the output 'Fred' is displayed in quotations
```

```
He said his name was Fred.
He said his name was 'Fred'.
```

As another example, `\t` inserts a tab into a string. Note that `%r` output the string representation and ignored the the slash `\` in `\t`, therefore a tab was not inserted.

[204]:
```
print('I once caught a fish %s.' %'this \tbig')
print('I once caught a fish %r.' %'this \tbig')
```

```
I once caught a fish this       big.
I once caught a fish 'this \tbig'.
```

The `%s` operator converts whatever it sees into a string, including integers and floats. Similarly, the `%d` operator converts numbers to integers. Note the difference below.

[205]:
```
print('I wrote %s programs today.' %3.75)
print('I wrote %d programs today.' %3.75)
```
```
I wrote 3.75 programs today.
I wrote 3 programs today.
```

Padding and Precision of Floating Point Numbers

Floating point numbers use the format `%5.2f`. Here, `5` is the minimum number of characters the string should contain; these characters may be padded with whitespace if the entire number does not have this many digits. Next to this, `.2f` stands for how many numbers to show past the decimal point.

[206]:
```
print('Floating point numbers: %5.2f' %(13.144))
```
```
Floating point numbers: 13.14
```

[207]:
```
print('Floating point numbers: %1.0f' %(13.144))
```
```
Floating point numbers: 13
```

[208]:
```
print('Floating point numbers: %1.5f' %(13.144))
```
```
Floating point numbers: 13.14400
```

[209]:
```
# Note that 5 empty spaces will be added in front of the number, to make it a total of␣
↪10 characters
print('Floating point numbers: %10.2f' %(13.144))
```
```
Floating point numbers:      13.14
```

[210]:
```
print('Floating point numbers: %25.2f' %(13.144))
```
```
Floating point numbers:                     13.14
```

It is possible to use more than one operator (e.g., `%s`, `%f`, and `%r`) in the same print statement.

[211]:
```
print('First: %s, Second: %5.2f, Third: %r' %('hi!',3.1415,'bye!'))
```
```
First: hi!, Second:  3.14, Third: 'bye!'
```

Formatting with the `.format()` Method

An improved way to format objects into strings for print statements is with the string `.format()` method. The syntax is:

```
'String here {} then also {}'.format('something1','something2')
```

For example:

[212]:
```
print('This is a string with an {}'.format('insert'))
```
```
This is a string with an insert
```

The `.format()` method has several advantages over the `%s` placeholder method:

1. Inserted objects can be called by index position.

[213]:
```
print('The {2} {1} {0}'.format('fox','brown','quick'))
```

```
The quick brown fox
```

2. Inserted objects can be assigned keywords.

[214]:
```
print('First Object: {a}, Second Object: {b}, Third Object: {c}'.format(a=1, b='Two',␣
↪c=12.3))
```

```
First Object: 1, Second Object: Two, Third Object: 12.3
```

3. Inserted objects can be reused, avoiding duplication.

[215]:
```
print('A %s saved is a %s earned.' %('penny','penny'))
# vs.
print('A {p} saved is a {p} earned.'.format(p='penny'))
```

```
A penny saved is a penny earned.
A penny saved is a penny earned.
```

Within the curly braces we can assign field lengths, left/right alignments, rounding parameters, and more.

[216]:
```
# The field 0 has a length of 8 characters, and the next field 1 has a length of 10␣
↪characters
print('{0:8} | {1:10}'.format('Fruit', 'Quantity'))
print('{0:8} | {1:10}'.format('Apples', 3.))
print('{0:8} | {1:10}'.format('Oranges', 10))
```

```
Fruit    | Quantity
Apples   |        3.0
Oranges  |         10
```

By default, `.format()` aligns text to the left, numbers to the right. We can pass an optional <,^, or > to set a left, center, or right alignment.

[217]:
```
print('{0:<8} | {1:^10} | {2:>8}'.format('Left','Center','Right'))
print('{0:<8} | {1:^10} | {2:>8}'.format(11,22,33))
```

```
Left     |   Center   |    Right
11       |     22     |       33
```

We can precede the alignment operator with a padding character.

[218]:
```
print('{0:=<8} | {1:-^10} | {2:.>8}'.format('Left','Center','Right'))
print('{0:=<8} | {1:-^10} | {2:.>8}'.format(11,22,33))
```

```
Left==== | --Center-- | ...Right
11====== | ----22---- | ...33
```

Field widths and float precision are handled in a way similar to placeholders. The following two print statements are equivalent.

[219]:
```
print('This is my ten-character, two-decimal number:%10.2f' %13.579)
print('This is my ten-character, two-decimal number:{0:10.2f}'.format(13.579))
```

```
This is my ten-character, two-decimal number:     13.58
This is my ten-character, two-decimal number:     13.58
```

Note that there are 5 spaces following `number:` in the output, and 5 characters taken up by 13.58, for a total of ten characters.

Formatting with String Literals (f-strings)

Introduced in Python 3.6, `f-strings` offer several benefits over the older `.format()` string method described above. E.g., we can bring outside variables immediately into the string rather than pass them as arguments through `.format(var)`.

[220]:
```
name = 'Fred'

print(f"He said his name is {name}.")
```

```
He said his name is Fred.
```

Pass `!r` to get the string representation.

[221]:
```
print(f"He said his name is {name!r}")
```

```
He said his name is 'Fred'
```

Float formatting follows the syntax `{value:{width}.{precision}}`.

Whereas with the `.format()` method we can write `{value:10.4f}`, with f-strings this becomes `{value:{10}.{6}}`.

[222]:
```
print("My 10 character, four decimal number is:{0:10.4f}".format(23.45678))
print(f"My 10 character, four decimal number is:{23.45678:{10}.{6}}")
print(f"My 10 character, two decimal number is:{23.45678:{10}.{4}}")
```

```
My 10 character, four decimal number is:   23.4568
My 10 character, four decimal number is:   23.4568
My 10 character, two decimal number is:     23.46
```

Note that with f-strings, *precision* refers to the total number of digits, not just those following the decimal. This fits more closely with scientific notation and statistical analysis. Unfortunately, f-strings do not pad to the right of the decimal, even if precision allows it.

[223]:
```
print(f"My 10 character, two decimal number is:{23.45:{10}.{4}}")
```

```
My 10 character, two decimal number is:     23.45
```

If this becomes important, we can always use `.format()` method syntax inside an `f-string`.

[224]:
```
print(f"My 10 character, four decimal number is:{23.45:10.4f}")
```

```
My 10 character, four decimal number is:   23.4500
```

7.2.10 References

1. Mark Lutz, "Learning Python," 5-th edition, O-Reilly, 2013. ISBN: 978-1-449-35573-9.
2. Pierian Data Inc., "Complete Python 3 Bootcamp," codes available at: https://github.com/Pierian-Data/Complete-Python-3-Bootcamp.
3. Course T81 558:Applications of Deep Neural Networks, Washington University in St. Louis, Instructor: Jeff Heaton, codes available at: https://github.com/jeffheaton/t81_558_deep_learning

BACK TO TOP

7.3 Lecture 3 - Statements, Files

7.3.1 3.1 Statements

Python code can be decomposed into **packages**, **modules**, **statements**, and **expressions**, as follows:

1. Packages are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. Expressions create and process objects.

Expressions are part of statements that return a value, such as variables, operators, or function calls.

For example, expressions in Python include `2+5`, or `x+3`, or `x * y` for given `x` and `y`, or `func1(3)` for a given function `func1`, since they all return a value. Expressions perform operations upon objects, and produce a value that can be used in other operations, can be assigned to a variable, printed out, etc. An expression can be part of larger expressions of statements. For instance, in the statement `a = 2 + 5`, the part `2 + 5` is an expression.

> **Statements** are sections of code that perform an action. The main groups of Python statements are: assignment statements, print statements, conditional statements (`if`, `break`, `continue`, `try`), and looping statements (`for`, `while`).

Therefore, `a = 2 + 5` is a statement, because it doesn't return a value, but it performs an action by assigning the value `5` to the variable `a`. Beside assignment statements (such as `a = 2 + 5`), other forms of statements include conditional statements (e.g., `if x > 5:`), looping statements (e.g., `for i in range(10):`), print statements (e.g., `print('hi')`), as well as there are several other statements, such as `import`, `return`, `pass`, etc. Note again that none of these statements returns a value in the way expressions do.

Modules are Python files that contain Python statements, and are also called scripts.

Thus, a module is a single Python script that is composed of a series of related statements grouped into one file. The statements in a module can assign values to variables, create functions or classes, or perform other actions.

Packages are Python programs that collect related modules together within a single directory hierarchy.

In other words, a package is a directory that contains a collection of module files. Packages can also contain sub-packages, forming a hierarchy of packages. A Python program can be organized into a single package, or more complex programs can use multiple packages to achieve its functionality.

In this lecture, we will study several types of Python statements. In a subsequent lecture, we will provide explanations about modules and packages in Python.

3.1.1 if, else, elif Statements

In Python, the `if` statement allows to instruct the program to perform alternative actions, based on one or several tests. This provides a means for introducing logic in our codes, and it can be interpreted as "if this case happens, then perform this action".

The `if` statement takes the form of an `if test`, which can be followed by one or more optional `elif (else if) tests`, and a final optional `else test`. Each of the tests has an associated block of nested statements, indented under a header line.

The `if` statement is a *compound statement,* since it may contain other statements (e.g., `elif` or `else`) in its syntax.

Compound statements in Python contain other statements, and they affect or control the execution of those other statements in some way. Compound statements typically span multiple lines.

Also, the `if` statement is referred to as a **conditional** statement, since it involves actions that are performed only when the conditions in the `if test` are satisfied.

Basic if Test

In its simplest form, the `if` statement has the following syntax:

```
if test1:
    code to execute when test1 is True
```

The `if` statement is used to perform a test and control whether or not the indented block of code is executed.

- The first line is a header line. It opens with the `if` test and ends with a colon `:` (omitting the colon at the end of `if` statement is one of the most common mistakes by beginner programmers in Python). The output of the expression in the `if` test is a *Boolean* variable (i.e., *True* or *False*).
- The block of code is indented under the header and contains one or more statements (such as the `print` statements in the next cell) that are executed if the test is *True*.

[1]:
```
x = 105

if x > 100:
    print(x, 'is high')
```

```
105 is high
```

[2]:
```
x = 105

# The print statement is not executed since x<50 is False
if x < 50:
    print(x, 'is high')
```

[3]:
```
y = 20

if y < 50:
    print (y, 'is low')
```

```
20 is low
```

[4]:
```
if True:
    print('It is true!')
```

```
It is true!
```

[5]:
```
# The print statement is not executed
if False:
    print('It is true!')
```

Therefore, the code indented under an `if` line will be executed only if the first line returns a Boolean `True` value. As we mentioned earlier, any nonzero number or nonempty array returns a Boolean `True`, and 0 or an empty array returns a `False`.

[6]:
```
if 1:
    print('It is true!')

if 5:
    print('It is also true!')

# The print statement is not executed
if 0:
    print('It is not true!')
```

```
It is true!
It is also true!
```

The if - else Tests

The `else` test allows to add additional logic to the `if` test.

Check the example in the following cell. Since we assigned a Boolean `False` to the variable `x`, the line `if x:` returns *False*, and as a result the statement indented under `if` will not be executed. In the case when the `if` test is *False*, the code after `else` is executed.

[7]:
```
x = False

if x:
    print('This is printed when x is True!')
else:
    print('This is printed when x is False')
```

```
This is printed when x is False
```

Here are more examples of using `else` to execute a block of code when an `if` test is *not* true.

[8]:
```
num = 43

if num > 100:
    print(num, 'is high')
else:
    print(num, 'is low')
```

```
43 is low
```

[9]:
```
num = 134

if num > 100:
    print(num, 'is high')
else:
    print(num, 'is low')
```

```
134 is high
```

In the next example we use the `input()` function to enter text using the keyboard (press the `Enter` key to confirm it).

[10]:
```
person = input("Enter your name: ")
# E.g., enter other name than Joe

if person == 'Joe':
    print('Welcome Joe!')
else:
    print("Welcome, Joe will be with you shortly?")
```

```
Enter your name:  Jim
```

```
Welcome, Joe will be with you shortly?
```

Note that:

- `else` is always attached to `if`, and it cannot be used as a standalone test.
- `else` allows to specify an alternative action to execute when the `if` test is `False`.

if - elif - else Tests

We can use `elif` to specify additional tests, when we want to provide several alternative cases, each with its own test. The statement `elif` is short for "else if" and it is always associated with an `if` statement. If there is an `else` test in the code, `elif` must come before `else`.

The general syntax is:

```
if test1:
    code to execute -> perform action 1
elif test2:
    code to execute -> perform action 2
else:
    code to execute -> perform action 3
```

The above compound statement can be interpreted as: if the case in test 1 happens, perform action 1. Else, if the case in test 2 happens, perform action 2. Or else, if none of the above cases happen, perform action 3.

That is, Python executes the statements nested under the `elif` test if the statements before the test are not *True*, and the statements under the `else` test are executed only when none of the `elif` tests is *True*.

Both the `elif` and `else` parts are optional and they may be omitted. As well as, there may be more than one `elif` statement nested in the `if` test.

The words `if`, `elif`, and `else` line up vertically, and need to have the same indentation. The block of code under each test needs to be aligned if it consists of multiple lines of code, however the code under `if` does not have to have the same indentation as the code under `elif` or `else` (although it is recommended to use consistent indentation).

[11]:
```
z = 68

if z > 100:
    print(z, 'is high')
elif z > 50:
    print(z, 'is medium')
else:
    print(z, 'is low')
```

```
68 is medium
```

[12]:
```
z = 30

if z > 100:
    print(z, 'is high')
elif z > 50:
    print(z, 'is medium')
else:
    print(z, 'is low')
```

```
30 is low
```

[13]:
```
location = 'Bank'

if location == 'Auto Shop':
    print('Welcome to the Auto Shop!')
elif location == 'Bank':
    print('Welcome to the bank!')
else:
    print('Where are you?')
```

```
Welcome to the bank!
```

Boolean Operators to Make Complex Statements

We can create complex conditional statements with Boolean operators like **and** and **or**, or use comparators like <, >, or other comparators.

[14]:
```
age = 40

if age > 65 or age < 16:
    print(age, 'is outside the labor force')
else:
    print(age, 'is in the labor force')
```

```
40 is in the labor force
```

We saw in the examples above that we can use double equal signs == to check if two objects are the same. Similarly, we can use an exclamation point and equal sign != to check if two objects are not the same.

[15]:
```
person = 'Jim'

if person != 'Joe':
    print("Welcome, what's your name?")
else:
    print('Welcome Joe!')
```

```
Welcome, what's your name?
```

The if - else Ternary Expression

Python also has an **if - else ternary expression** with the following syntax:

```
a if condition else b
```

In the above expression, first the `condition` is evaluated, and afterward either `a` or `b` is returned based on the Boolean value of the condition.

Let's reconsider the `if-else` statement example that we saw earlier.

[16]:
```
num = 43

if num > 100:
    print(num, 'is high')
else:
    print(num, 'is low')
```

```
43 is low
```

The corresponding `if-else` ternary expression is as follows.

[17]:
```
print(num, 'is high') if num > 100 else print(num, 'is low')
```

```
43 is low
```

The ternary expression allows to reduce the above 4 lines of code into 1 line. Based on the value of the condition `num > 100`, if the condition is True then `print(num, 'is high')` is executed, and if the condition is False then `print(num, 'is low')` is executed.

Handling Case Switch

If you used languages like C, Pascal, or MATLAB, and if you are interested to know if there is a *switch* or *case* statement in Python that selects an action based on a variable's value, there isn't. Instead, in Python we can code multiway branching as a series of if-elif tests.

An example is shown below. Note again that we can use as many `elif` statements as we want, but there can be only one `else` statement.

[18]:
```
choice = 'ham'

if choice == 'spam':
    print(2.25)
elif choice == 'ham':
    print(1.75)
elif choice == 'eggs':
    print(0.75)
elif choice == 'bacon':
    print(1.10)
else:
    print('Bad choice')
```

```
1.75
```

Although, it may be more convenient to create a dictionary to handle case switching instead of `if-elif-else` especially when there are many cases involved.

[19]:
```
branch = {'spam': 2.25, 'ham': 1.75, 'eggs': 0.75, 'bacon': 1.10}

choice = 'eggs'

if choice in branch:
    print(branch[choice])
else:
    print('Bad choice')
```

```
0.75
```

Indentation Rules

Python uses indentation of statements under a header to group the statements in a nested block. In the figure below, there are 3 blocks of code, each having a header line. Note that Block 1 is nested under Block 0, and it is indented further to the right of Block 0. Then, Block 2 is nested under Block 1, and it is intended even further to the right of Block 1.

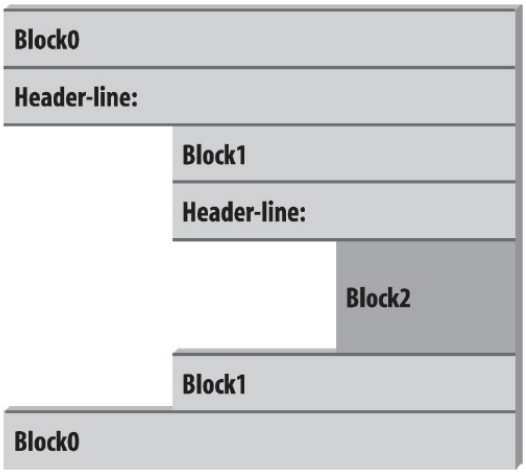


Figure source: Reference [1].

The indentation in Python is used to detect block boundaries. All statements indented the same distance to the right belong to the same block of code. The block ends either when a less-indented line or the end of the file is encountered.

Indentation may consist of any number of spaces, but it must be the same for all the statements in a single block. Four spaces or one tab per indentation level are commonly used, but there is no absolute standard for the number of spaces in indentation. However, it is not recommended to mix spaces and tabs for indentation within a block, because such indentation may look different in other editors and the codes can be more difficult to edit.

Look at the example in the next cell. It contains three blocks: the first block (Block 0, `if x:`) is not indented at all, the second (Block 1, `y = 2`) is indented four spaces under Block 0, and the third (Block 2, `print ('Block 2')` is indented eight spaces.

[20]:
```
x = 1
if x:
    y = 2
    if y:
        print('Block 2')
    print('Block 1')
print('Block 0')
```

```
Block 2
Block 1
Block 0
```

Several common mistakes with code indentation are shown below, which result in errors.

[22]:
```
x = 1
  if x: # Error: first line indented, this line belongs to Block 0 and it shouldn't be␣
↪indented
    y = 2
      if y:  # Error: unexpected indentation, this line should have the same indentation␣
↪as 'y = 2'
        print('Block 2')
   print('Block 1') # Error: inconsistent indentation, this line is indented 3 spaces,␣
↪and 'y = 2' is indented 4 spaces
print('Block 0')
```

```
  File <tokenize>:6
    print('Block 1') # Error: inconsistent indentation, this line is indented 3 spaces,␣
↪and 'y = 2' is indented 4 spaces
```

(continues on next page)

(continued from previous page)

```
    ^
IndentationError: unindent does not match any outer indentation level
```

To indent several lines of code for one tab, select the lines and then press either the `Tab` key or press the keys `Ctrl` + `]`. To unindent several lines of codes for one tab, press the keys `Ctrl` + `[`.

Statement Delimiters: Lines and Continuations

Python expects `if` statements to be written on a single line.

The code below produces an error because the `if` statement spans on two lines.

[23]:
```
num = 80

if num > 20 and num > 50 and
    num < 200 and num < 100:
    print('Medium number')
```

```
  Cell In[23], line 3
    if num > 20 and num > 50 and
                                ^
SyntaxError: invalid syntax
```

When a statement is too long to fit on a single line, there are two ways to make it span multiple lines.

The first one is to enclose the statement either in a pair of parentheses `()`, square brackets `[]`, or curly braces `{}`. Continuation lines do not need to be indented at any level, but it is a good practice to align the lines vertically for readability.

Examples are shown below.

[24]:
```
num = 80

if (num > 20 and num > 50 and
    num < 200 and num < 100):
    print('Medium number')
```

```
Medium number
```

[25]:
```
# Note that the indentation is not required for continuation lines enclosed in a pair of␣
↪parentheses, brackets, or braces
num = 80
if {num > 20 and num > 50 and
  num < 200 and num < 100}:
    print('Medium number')
```

```
Medium number
```

[26]:
```
L = ["Good",
"Bad",
"Ugly"]
```

(continues on next page)

(continued from previous page)

```
L
```

```
['Good', 'Bad', 'Ugly']
```

Also, statements can span multiple lines if they end in a backward slash \. Although, this is an older feature, and it is not generally recommended. One reason is because if there are empty spaces after the backward slash, it will result in an error.

```
num = 80
if num > 20 and num > 50 and \
    num < 200 and \
    num < 100:
    print('Medium number')
```

```
Medium number
```

The above line continuation rules apply to any other statements and expressions.

```
x = 1 + 2 + 3 \
+4
x
```

```
10
```

```
x = (1 + 2 + 3
+4)
x
```

```
10
```

Note also that Python allows to write more than one noncompound statement (i.e., statements without nested statements) on the same line, separated by semicolons.

```
x = 5; print(x)
```

```
5
```

Python allows to write the body of a compound statement (like `if`) on the same line with the header, provided the body contains just simple (noncompound) statements (i.e., without `elif` or `else` tests).

```
if True: print('Something')
```

```
Something
```

3.1.2 for Loops

A `for` loop acts as an iterator in Python. It goes through items that are in a *sequence* or any other iterable object. Objects that we've learned about that we can iterate over include strings, lists, and tuples. And even dictionaries allow to iterate over keys or values.

The general format of a `for` loop in Python is:

```
for item in object:
    code to execute -> perform actions
```

The variable name used for the `item` is completely up to the coder, so use your best judgment for choosing a name that makes sense and you will be able to understand when revisiting your code. This `item` can then be referenced inside your loop, for example if you wanted to use if statements to perform checks.

[32]:
```
list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

[33]:
```
for num in list1:
    print(num)
```

```
1
2
3
4
5
6
7
8
9
10
```

Add an if statement to check for even numbers.

[34]:
```
for num in list1:
    if num % 2 == 0:
        print(num)
```

```
2
4
6
8
10
```

We could have also included an `else` statement.

[35]:
```
for num in list1:
    if num % 2 == 0:
        print(num)
    else:
        print('Odd number')
```

```
Odd number
2
Odd number
4
Odd number
6
Odd number
8
Odd number
10
```

Another common practice with `for` loops is to keep some sort of running tally during multiple loops. For example, let's create a `for` loop that sums up the elements in a list.

[36]:
```
# Start sum at zero
list_sum = 0

for num in list1:
    list_sum = list_sum + num

print(list_sum)
```

```
55
```

Or, we could have used the operator += to perform the addition towards the sum.

[37]:
```
# Start sum at zero
list_sum = 0

for num in list1:
    list_sum += num

print(list_sum)
```

```
55
```

We can also use `for` loops with strings and tuples, since they are sequences, so when we iterate through them we will be accessing each item in the sequence.

[38]:
```
for letter in 'This is a string.':
    print(letter)
```

```
T
h
i
s

i
s

a

s
t
r
i
n
g
.
```

[39]:
```
# loop through a dictionary
d = {'k1':1, 'k2':2, 'k3':3}
```

[40]:
```
for item in d:
    print(item)
```

```
k1
k2
k3
```

Notice how the above produces only the keys.

We can also use the dictionary methods **.keys()**, **.values()**, and **.items()** with `for` loops. In Python each of these methods returns a *dictionary view object*. The view objects provide a view of the dictionary's key, values, and items (pairs of keys and values). The dictionary view objects support operations like membership tests and iterations over the keys, values, and items. The type of the the view objects is `dict_items`. If we make changes to the dictionary, the view objects will keep track of the changes.

[41]:
```
# Create a dictionary view object
d.items()
```

[41]:
```
dict_items([('k1', 1), ('k2', 2), ('k3', 3)])
```

Since the `.items()` method supports iteration, we can print both the keys and values.

[42]:
```
# Dictionary unpacking
for k,v in d.items():
    print(k)
    print(v)
```

```
k1
1
k2
2
k3
3
```

If we want to obtain a list of keys, values, or key-value tuples, we can *cast* the dictionary view objects as a list.

[43]:
```
list(d.keys())
```

[43]:
```
['k1', 'k2', 'k3']
```

[44]:
```
# Compare to
d.keys()
```

[44]:
```
dict_keys(['k1', 'k2', 'k3'])
```

[45]:
```
list(d.values())
```

[45]:
```
[1, 2, 3]
```

[46]:
```
list(d.items())
```

[46]:
```
[('k1', 1), ('k2', 2), ('k3', 3)]
```

Another used function is **range** which allows to quickly *generate* a list of integers, and it is often used with `for` loops.

[47]:
```
string = 'abcde'

n = len(string)
for i in range(n): # i is the index
    print('Index', i, 'Letter', string[i])
```

```
Index 0 Letter a
Index 1 Letter b
Index 2 Letter c
```

(continues on next page)

(continued from previous page)

```
Index 3 Letter d
Index 4 Letter e
```

In general, `range` can have 3 parameters to pass: a start, a stop, and a step size. Let's see some examples.

[48]:
```
# To get a list when using range, we need to cast it to a list
# Parameters: start, stop, step size
list(range(0, 101, 10))
```

[48]:
```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

[49]:
```
# Default step size is 1
# Notice that 11 is not included
list(range(0, 11))
```

[49]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

[50]:
```
# Default start is 0
list(range(6))
```

[50]:
```
[0, 1, 2, 3, 4, 5]
```

The **enumerate** function is another useful function to use with `for` loops. It returns both the index and the item in each loop.

[51]:
```
string = 'abcde'

for i,letter in enumerate(string):
    print('Index', i,'Letter:', letter)
```

```
Index 0 Letter: a
Index 1 Letter: b
Index 2 Letter: c
Index 3 Letter: d
Index 4 Letter: e
```

3.1.3 while Loops

The `while` statement in Python is another way to perform iteration. A `while` statement will repeatedly execute a single statement or group of statements as long as the condition is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

The general format of a `while` loop is:

```
while test:
    code to execute -> perform action 1
else:
    code to execute -> perform action 2
```

Let's look at a few simple while loops in action.

[52]:
```
x = 0
```

(continues on next page)

(continued from previous page)

```
while x < 5:
    print('x is currently: ', x)
    print('x is still less than 5, adding 1 to x')
    x+=1
```

```
x is currently:  0
x is still less than 5, adding 1 to x
x is currently:  1
x is still less than 5, adding 1 to x
x is currently:  2
x is still less than 5, adding 1 to x
x is currently:  3
x is still less than 5, adding 1 to x
x is currently:  4
x is still less than 5, adding 1 to x
```

We can also add an else statement:

[53]:
```
x = 0

while x < 5:
    print('x is currently: ',x)
    print(' x is still less than 5, adding 1 to x')
    x+=1
else:
    print('All Done!')
```

```
x is currently:  0
 x is still less than 5, adding 1 to x
x is currently:  1
 x is still less than 5, adding 1 to x
x is currently:  2
 x is still less than 5, adding 1 to x
x is currently:  3
 x is still less than 5, adding 1 to x
x is currently:  4
 x is still less than 5, adding 1 to x
All Done!
```

[ ]:
```
## DO NOT RUN THIS CODE!!!!
# while True:
#    print("I'm stuck in an infinite loop!")
```

A quick note: If you *did* run the above cell, click on the Kernel menu above to restart the kernel!

3.1.4 break, continue, pass Statements

We can use `break`, `continue`, and `pass` statements in our loops to add additional functionality for various cases.

With the `break` and `continue` statements, the general format of the `while` loop looks like this:

```
while test:
    code to execute -> perform action 1
    if test:
        break         # Exit the 'while' loop now
        continue      # Skip the 'else' statements (if present) and go to top of the
↪'while' loop now
    else:
        code to execute -> perform action 2   # Run these statements when the 'if' test␣
↪is False
```

The `break` and `continue` statements can appear anywhere inside the loop's body, but they are usually nested in an `if` statement to perform an action based on some condition.

[54]:
```
for letter in "string":
    if letter == "i":
        break # exit the 'for' loop now
    print(letter)

print("The end")
```

```
s
t
r
The end
```

[55]:
```
for letter in "string":
    if letter == "i":
        continue # go to the top of the 'for' loop now (skip the commands following
↪'continue')
    print(letter)

print("The end")
```

```
s
t
r
n
g
The end
```

Two more examples follow with an `else` statement.

[56]:
```
x = 0

while x < 5:
    print('x is currently: ', x)
    print(' x is still less than 5, adding 1 to x')
    x += 1
    if x == 3:
```

(continues on next page)

(continued from previous page)

```
        print('Breaking because x == 3')
        break  # terminate the 'while' loop, go to the 'print('The end')' statement
    else:
        print('continuing...')

print('The end')
```

```
x is currently:  0
 x is still less than 5, adding 1 to x
continuing...
x is currently:  1
 x is still less than 5, adding 1 to x
continuing...
x is currently:  2
 x is still less than 5, adding 1 to x
Breaking because x == 3
The end
```

[57]:
```
x=0

while x < 5:
    print('x is currently: ', x)
    print(' x is still less than 5, adding 1 to x')
    x += 1
    if x == 3:
        print('Continuing to the next step')
        continue  # Skip the rest of the lines, and go to the while loop
        print('This line will be skipped and will not be printed')
    else:
        print('continuing...')

print('The end')
```

```
x is currently:  0
 x is still less than 5, adding 1 to x
continuing...
x is currently:  1
 x is still less than 5, adding 1 to x
continuing...
x is currently:  2
 x is still less than 5, adding 1 to x
Continuing to the next step
x is currently:  3
 x is still less than 5, adding 1 to x
continuing...
x is currently:  4
 x is still less than 5, adding 1 to x
continuing...
The end
```

The statement `pass` is generally used as a placeholder and it does not do anything. Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body, because this would give an error. So, we use the `pass` statement to construct a body that does nothing.

[58]:
```
# Pass is just a placeholder for functionality to be added later
sequence = {'p', 'a', 's', 's'}
for val in sequence:
    pass
```

[59]:
```
# Pass can be used as a placeholder for a function or a class
def my_function(arguments):
    pass

class Example:
    pass
```

7.3.2 3.2 Files

Python uses **file objects** to interact with external files on your computer. These file objects can be any sort of file you have on your computer, such as a text file, Excel document, email, audio file, picture, etc.

Python has a built-in `open()` function that allows us to open and write to files.

The `open()` function requires to pass two arguments: filename and processing mode. The **filename** is simply the name of the file, and for reading it, it is assumed that the file exists in the current working directory: if that is not the case, the `filename` should also include the path to the file. The **processing mode** can be either the string `'r'` to read the file, `'w'` to write to the file (create a file and open it for writing), or `'a'` to append text to an existing file. Also, adding `+` to the mode allows to both read and write to a file (e.g., `'r+'`, `'a+'`). Both the filename and processing mode should be strings.

```
afile = open(filename, mode)
```

The `open` function creates a Python **file object** named `afile` in this example (we can select any valid name for it), which serves as a link to the file residing on the computer named `filename`. The file object allows to transfer strings of data to and from the linked file `filename`.

Writing to a File

For example, let's create a simple text file called `test.txt` having two lines of text. The `open` function in the example below will return a file object named `myfile`, which has a `write()` method for data transfer.

The file `test.txt` will be saved in the current working directory.

[60]:
```
# Create an empty file
myfile = open('test.txt','w')
```

The `write(string)` method of the file object named `myfile` allows to write a string to the file. In the next example, the string `'hello text file\n'` is written. Note also that we need to include the end-of-line terminator `\n` in the string, otherwise the next `write` command will continue the current line.

[61]:
```
# Write a line of text: string
myfile.write('hello text file\n')
# Note that the write call returns the number of characters in the string
```

[61]:
```
16
```

[62]:
```
myfile.write('goodbye text file\n')
```

[62]:
```
18
```

[63]:
```
myfile.close()
```

Now, click on the `test.txt` file in the Jupyter Lab dashboard, to inspect if it looks as we expect.

Use caution when opening an existing file for writing with `w`, as it truncates the original file, meaning that any existing content in the original file is deleted. Let's try the following code.

[64]:
```
myfile = open('test.txt','w')
myfile.write('This is a first line\n')
myfile.write('This is a second line\n')
myfile.close()
```

Now open the file `test.txt` and you will notice that it has been overwritten.

Opening a file

Let's open the file `test.txt`.

[65]:
```
# Open for text input: 'r' is default mode and it can be omitted
myfile = open('test.txt','r')
```

[66]:
```
# Read the lines one at a time
myfile.readline()
```

[66]:
```
'This is a first line\n'
```

[67]:
```
# Read the lines one at a time
myfile.readline()
```

[67]:
```
'This is a second line\n'
```

[68]:
```
# Empty string: end-of-file (EOF)
myfile.readline()
```

[68]:
```
''
```

In addition, using the `read` method we can read the entire file into a string all at once.

[69]:
```
myfile = open('test.txt')
myfile.read()
```

[69]:
```
'This is a first line\nThis is a second line\n'
```

Or, if we use `print` the content will be displayed in a readable format without showing the `\n`characters.

[70]:
```
myfile = open('test.txt')
print(myfile.read())
```

```
This is a first line
This is a second line
```

Also note that we can write the above cell into one single line:

[71]:
```
print(open('test.txt').read())
```

```
This is a first line
This is a second line
```

One confusing thing about reading files is that if we try to read the same file object twice, we'll find out that it only gets read once.

[72]:
```
myfile = open('test.txt')
myfile.read()
```

[72]:
```
'This is a first line\nThis is a second line\n'
```

[73]:
```
# What happens if we try to read the file again?
myfile.read()
```

[73]:
```
''
```

This happens because file objects remember their position, and after we read the file the first time, the reading 'cursor' was at the end of the file, and there was nothing left to read.

We can reset the 'cursor' like this:

[74]:
```
# Seek to the start of file (index 0)
myfile.seek(0)
```

[74]:
```
0
```

[75]:
```
# Now read again
myfile.read()
```

[75]:
```
'This is a first line\nThis is a second line\n'
```

When we have finished using the file, it is always good practice to close it.

[76]:
```
myfile.close()
```

You can also sometimes see another code variant, where `open` is used within a `with` statement, like in the example shown below. An advantage of this approach is that the `with` statement automatically closes the file after the block, as well as it makes handling any unexpected errors easier. Thus, it is the preferred way for opening files by many Python users.

[77]:
```
with open('test.txt', 'r') as myfile:
    data = myfile.read()

print(data)
```

```
This is a first line
This is a second line
```

Alternatively, to read files from other directories on your computer (instead of the current working directory), enter the entire file path.

For Windows, one option is to use double backslashes `\\`, so that Python doesn't treat the second `\` as part of an escape character (such as `\n`, `\t`, etc.):

```
myfile = open('C:\\Users\\YourUserName\\Desktop\\MyFolder\\test.txt')
```

For example, note that the \n escape character in the following cell introduces an unwanted new line.

[78]:
```
print('C:\some\name')
```

```
C:\some
ame
```

This is corrected by using double backslashes.

[79]:
```
print('C:\\some\\name')
```

```
C:\some\name
```

For Mac OS and Linux, use forward slashes.

```
myfile = open('/Users/YourUserName/MyFolder/test.txt')
```

In latest Python versions `open` works with either forward slashes or backward slashes, so either is fine. However, the problem with the single and double slashes in the examples above is that codes written on a Windows machine will not work on Unix machines, and vice versa. Therefore, a preferred option for Windows would be to use a `raw string` and single backslashes as shown below.

```
myfile = open(r'C:\Users\YourUserName\Desktop\MyFolder\test.txt')
```

The `raw string` form (use of `r` before the string) turns off escape characters in strings.

Note that `C:\Users\YourUserName\Desktop\MyFolder\test.txt` is an **absolute path** because it lists all directories on the disk `C:` to access the file `test.txt`. The path can also be a **relative path**, where for example if we are currently in a current working directory `C:\Users\YourUserName\Desktop` we can use `MyFolder\test.txt` as a path for the `filename` relative to the current working directory.

Appending to a File

Passing the argument `'a'` as a processing mode opens the file and puts the pointer at the end for appending. Similarly, `'a+'` allows us to both read and write to a file. If the file does not exist, one will be created.

[80]:
```
myfile = open('test.txt','a+')
myfile.write('\nThis is text being appended to test.txt\n')
myfile.write('And another line here\n')
```

[80]:
```
22
```

[81]:
```
myfile.seek(0)
print(myfile.read())
```

```
This is a first line
This is a second line

This is text being appended to test.txt
And another line here
```

[82]:
```
myfile.close()
```

Iterating through a File

When reading a file line by line, the entire file is held in the memory. Using file iterators, such as a `for` loop, is often preferred with large files. The created file object by `open` will automatically read and return one line on each loop iteration.

[83]:
```
for line in open('test.txt'):
    print(line)
```

```
This is a first line

This is a second line



This is text being appended to test.txt

And another line here
```

Reading and Writing Binary Files

In the above sections, we used the `open()` function in *text mode*, which allows to read and write strings from and to files. The `open()` function can also be used in *binary mode* that allows to read and write binary files. This mode is useful for working with non-textual files in Python, such as images, audio files, compressed files, etc. For reading and writing binary files, the *processing mode* in the `open()` function should be set to `'rb'` to read the file, and `'wb'` to write to the file, where the added letter `b` indicates that the function is applied to processing binary files.

When reading a file in binary mode, Python will read every byte in the file as is, and return a *byte string*. Conversely, in text mode, Python will decode the information in the file into text characters, and return a *text string*.

[84]:
```
image_file = open('images/house.png', 'rb')
image_content = image_file.read()
image_file.close()
```

[85]:
```
type(image_content)
```

[85]:
```
bytes
```

Note however that there are other Python packages that provide advanced functionalities for working with non-textual files, in comparison to the `open()` function. For instance, for working with image files, the Python libraries *OpenCV*, *Pillow*, *ImageIO* are almost always preferred by the users.

Storing Python Objects in Files: Conversions

Let's next consider an example where multiple Python objects are written into a text file on multiple lines. The objects need to be first converted to strings, as the `write` method does not do any automatic to-string formatting.

[86]:
```
# Introduce numbers, string, dictionary, and list objects
S = 'Spam'
X, Y, Z = 43, 44, 45
D = {'a': 1, 'b': 2}
L = [1, 2, 3]

# Create output text file
F = open('datafile.txt', 'w')
# The lines in the string variable S above should end with \n
F.write(S + '\n')
# Convert numbers to strings
F.write('%s,%s,%s\n' % (X, Y, Z))
# Convert to strings and separate wtih \n
F.write(str(L) + '\n' + str(D) + '\n')
F.close()
```

Next, let's open the file and read it.

Notice in the next two cells that the displayed output gives the raw string content, while the `print` operation interprets the embedded end-of-line characters to render a formatted display.

[87]:
```
content = open('datafile.txt').read()
# String display
content
```

[87]:
```
"Spam\n43,44,45\n[1, 2, 3]\n{'a': 1, 'b': 2}\n"
```

[88]:
```
# User-friendly display
print(content)
```

```
Spam
43,44,45
[1, 2, 3]
{'a': 1, 'b': 2}

```

To convert the strings in the text file into Python objects, we will need to use conversion tools.

For instance, `rstrip()` removes the end-of-line character `\n`.

[89]:
```
# Open the file again, this time using the object named F
F = open('datafile.txt')
# Read the first line (see above)
line = F.readline()
line
```

[89]:
```
'Spam\n'
```

[90]:
```
# Remove end-of-line
s = line.rstrip()
s
```

[90]:
```
'Spam'
```

The next line contains the string of numbers `'43,44,45\n'`, for which `split()` can be used to separate the numbers.

[91]:
```
# Next line from file
line = F.readline()
line
```

[91]:
```
'43,44,45\n'
```

[92]:
```
# Split on commas
parts = line.split(',')
parts
```

[92]:
```
['43', '44', '45\n']
```

[93]:
```
# int() converts to integer numbers
numbers = [int(P) for P in parts]
numbers
```

[93]:
```
[43, 44, 45]
```

[94]:
```
x, y, z = numbers
x, y, z
```

[94]:
```
(43, 44, 45)
```

To covert the list and dictionary we will use `eval()` which treats a string as executable code containing a Python expression.

[95]:
```
line = F.readline()
line
```

[95]:
```
'[1, 2, 3]\n'
```

[96]:
```
line.rstrip()
```

[96]:
```
'[1, 2, 3]'
```

[97]:
```
l = eval(line)
l
```

[97]:
```
[1, 2, 3]
```

[98]:
```
type(l)
```

[98]:
```
list
```

[99]:
```
line = F.readline()
line
```

[99]:
```
"{'a': 1, 'b': 2}\n"
```

[100]:
```
d = eval(line)
d
```

[100]:
```
{'a': 1, 'b': 2}
```

[101]:
```
type(d)
```

[101]:
```
dict
```

The above process of converting strings to Python objects is time-consuming and tedious, even for this simple example. Fortunately, there are simpler ways to write and read files in Python, which do not require the above conversion steps. Next, we will learn about the Python built-in modules `pickle` and `JSON` for storing Python objects, and in another lecture we will learn about the library `pandas` for reading and writing to files.

Storing Python Objects with pickle

Python's module `pickle` allows storing almost any Python object in a file directly, without the requirement for conversions to and from strings. To store the above list `L` in a file, we can pickle it directly by using the method `pickle.dump()`.

[102]:
```
import pickle
F = open('newdatafile.pkl', 'wb') # 'wb' used for writing to a binary file, indicates␣
↪that the content of the file is not text
# Pickle any object to file
pickle.dump(L, F)
F.close()
```

Then, to read the file and get the list, we simply use pickle again (a.k.a. unpickling) via the method `pickle.load()`.

[103]:
```
# Load any object from file
F = open('newdatafile.pkl', 'rb') # similarly, 'rb' used for reading a binary file
list1 = pickle.load(F)
list1
```

[103]:
```
[1, 2, 3]
```

[104]:
```
F.close()
```

The `pickle` module performs conversion of Python objects to string representation, referred to as **object serialization**, and reverse conversion of strings to Python objects, which is called **object deserialization**.

Storing Python Objects with JSON

`JSON` (stands for JavaScript Object Notation) is a newer data interchange format, which allows using stored data across programming languages (unlike `pickle` which works only with Python). On the other hand, `JSON` does not support as broad range of Python object types as `pickle`.

The following example shows translating the above dictionary `D` into `JSON` format to be saved into a file, and recreating the dictionary from the `JSON` format when it is loaded from the file.

[105]:
```
import json
FJ = open('json_datafile.text', 'w')
# Serialize the dictionary to a text file
json.dump(D, FJ)
FJ.close()
```

[106]:
```
# Deserialize the text file to a dictionary
new_d = json.load(open('json_datafile.text'))
new_d
```

[106]:
```
{'a': 1, 'b': 2}
```

7.3.3 Appendix: Python Interpreter

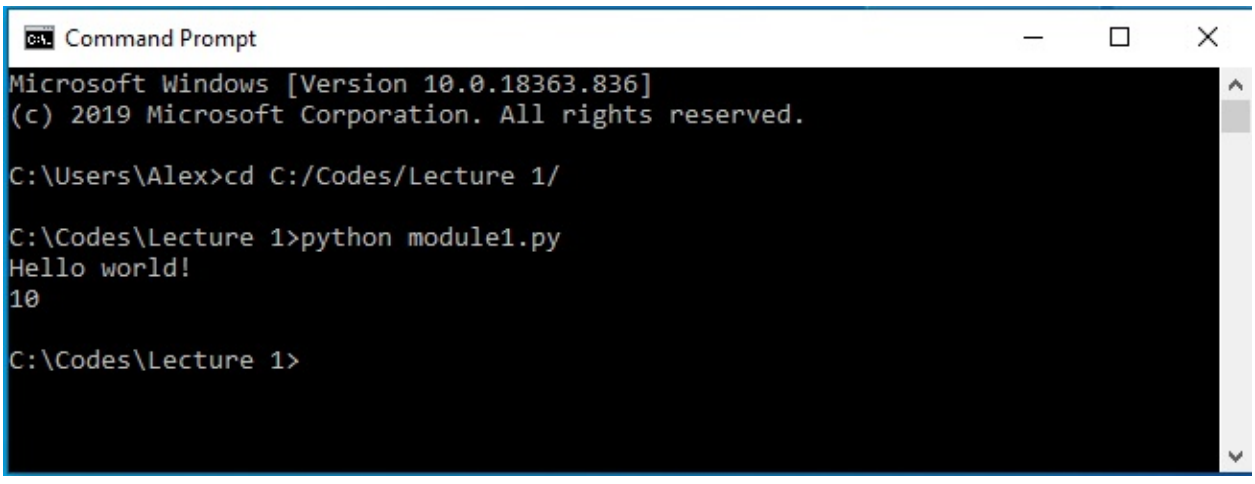
The material in the Appendix is not required for quizzes and assignments.

The **interpreter** in Python is the program that executes other programs. When you run your programs in Python, the interpreter reads your programs, and carries out the instructions contained in the program. Or, we can say that the interpreter interprets your codes and enables the hardware on your computer to execute the program.

When you install Python on your computer, the Python interpreter will be part of the installation, either as an executable program, or as a set of linked libraries. Note that there are several different Python installations, and depending on the type of Python installation you have on your computer, the interpreter may be implemented as a C program, a set of Java classes, or in another programming language.

Understanding how the programs are executed in Python can be helpful for programmers. For instance, I saved the following simple file as *module1.py*.

When I run the file in the Command Prompt, Python executed the file, and the output of the program is `Hello world!` and `10`.



When we run programs in Python, the programs are first compiled into **byte code**, and are afterward run by a **Python virtual machine (PVM)**, as shown in the figure below.

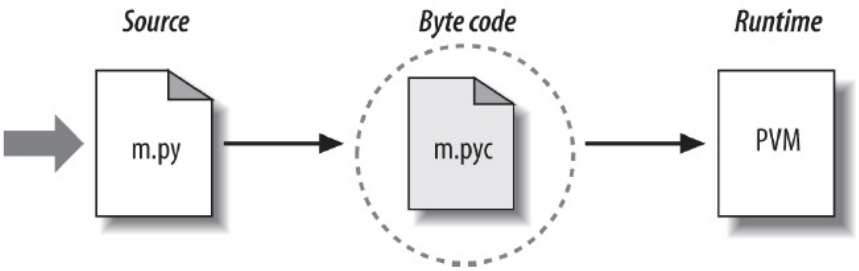
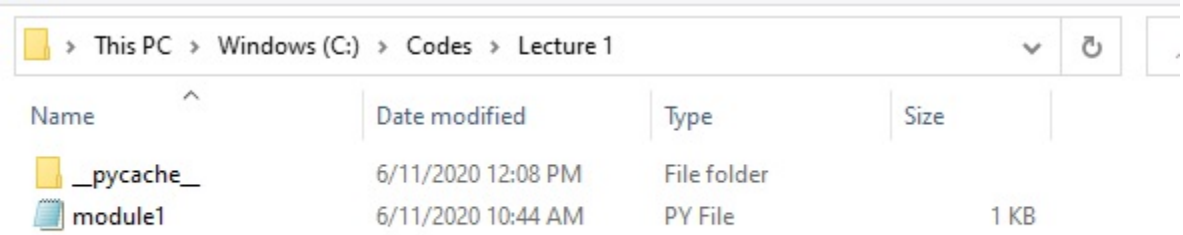


Figure source: Reference [1].

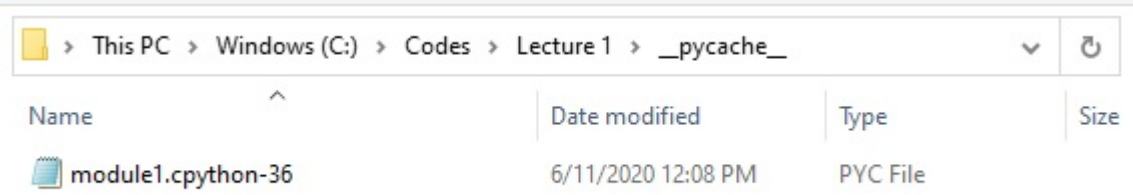
Byte code is a format into which the **source code** (the statements in the file) is compiled by the Python interpreter. Byte-code is platform-independent (i.e., it can be run on Windows, Linux, MacOS), and it can be run more quickly than the source code program.

The byte code is stored in a file with a *.pyc* extension, which stands for compiled .py file. The *.pyc* files are saved in a subdirectory named __pycache__ located in the same directory where the source file is saved.

For example, the directory where *module1.py* is saved on my computer is shown below, and the __pycache__ subdirectory was automatically created by Python.



Within the subdirectory is the byte code file named *module1.cpython-36*. The name indicates that the Python installation on my computer uses the CPython interpreter, and the installed Python version is 3.6. Note that the file type is PYC file, meaning a .pyc extension.



Byte code is saved for speed optimization. When I run module1.py next time, Python will skip the compilation step, and it will directly load the saved .pyc byte code file. However, if the original source code file module1.py was modified, Python will re-compile and update the byte code file. Similarly, if a different version of Python is installed, a new byte code file will be created that matches the current version of Python.

Python virtual machine (PVM) is the last part of the Python interpreter. PVM executes the byte code instructions one-by-one, i.e., it is the component that runs the programs. PVM is not a separate program, and it does not need to be installed separately: it is part of the Python installation. PVM needs a software layer to allocate physical computing resources—such as processors, memory, and storage.

Python belongs to the group of **interpreted languages**, or they are also called *scripting languages* (other languages in this group are Perl, Ruby, and JavaScript). As we explained, the Python interpreter reads the statements in source files and converts them into byte code files, which are afterwards executed by the PVM. Conversely, Java, C, and C++

belong to the group of **compiled languages**. In these languages, a compiler converts the statements in source files into binary machine code, which is afterwards executed by the computer hardware. Note that byte code files are different than binary machine code files. Consequently, running Python programs is slower than running C or C++ programs, because the code is interpreted as it is executed. On the other hand, writing and testing Python programs is faster and easier than writing and testing programs using compiled languages. (One last clarification: Python does compile source files, but the result is not a binary machine code, and because of that it is not considered a compiled language).

As we mentioned earlier, there are several different implementations of the Python interpreter. They include CPython, Jython, IronPython, Stackless Python, and PyPy. CPython is the standard, original implementation of Python, Jython is a Python implementation targeted for integration with the Java programming language, IronPython was designed to allow Python programs to integrate with applications coded to work with Microsoft's .NET Framework for Windows, etc.

7.3.4 References

1. Mark Lutz, "Learning Python," 5-th edition, O-Reilly, 2013. ISBN: 978-1-449-35573-9.
2. Pierian Data Inc., "Complete Python 3 Bootcamp," codes available at: https://github.com/Pierian-Data/Complete-Python-3-Bootcamp.

BACK TO TOP

7.4 Lecture 4 - Functions, Iterators, Generators

7.4.1 4.1 Functions

Functions are one of the main building blocks in Python allowing to construct and reuse code, without the need to repeatedly write the same code again and again. A function is a self-contained block of code that encapsulates a specific task or related group of tasks. The code defines the relationships between the inputs and the outputs of the function. The input arguments are passed when the function is called, afterward the program executes the code, and returns the output of the function.

Functions in programming languages are similar to mathematical functions that define a relationship between one or more inputs and outputs. For instance, the mathematical function represented with `z = f(x, y)` maps the inputs `x` and `y` into an output `z`. In programming languages, functions also operate over inputs and produce outputs, however, programming functions are much more generalized and versatile than mathematical functions, as they can operate over different objects and data types and can perform a wide variety of operations over the inputs.

4.1.1 Function Definition and Function Call

The basic syntax of functions in Python has the following form:

```
def name_of_function(argument1, argument2, ...):    # header line
    '''
    Optional Document String (docstring)
    This is where the function's docstring goes.
    When help(name_of_function) is called,
    this section will be printed out.
    '''
    # Define actions to perform
    # Return outputs (optional)
```

The syntax of functions begins with the `def` keyword, which informs Python that a new function is defined. The keyword `def` is followed by a space and the **name of the function**. Try to keep function names relevant, and avoid using function names that have the same names as built-in functions in Python (such as `len`, `str`, `print`, etc.).

The **arguments** of the function are written inside a pair of parentheses `()` in the header line and are separated by a comma. The arguments are the **input parameters** for the function.

At the end of the Python function header there is a colon punctuation mark `:`.

The code under the header line needs to be indented properly, by following the indentation rules which we mentioned in the previous lecture.

The **docstring** section is optional, and it should contain a basic description of the function. Although docstrings are not necessary for simple functions, it is good practice to write them for more complex functions, so that other people can easily understand our code.

After the docstring follows the body of the function which contains code block of statements that define the **actions** the function should perform.

Lastly, the function can `return` desired results as **outputs**.

The following figure illustrates the elements of a simple function.

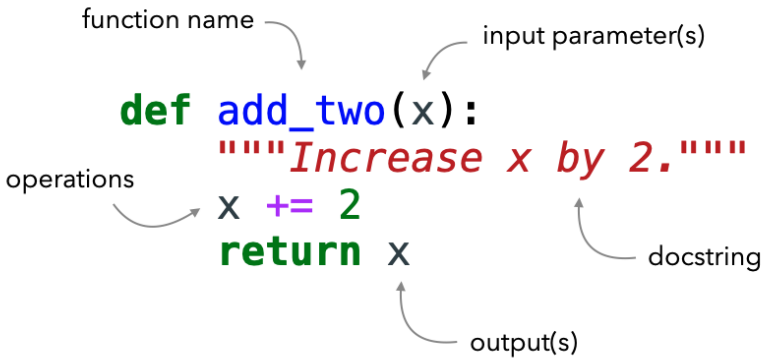


Figure source: Reference [4].

One very simple example of a function is shown next. In the header line, the `def` keyword indicates that a function `say_hello` is being defined. Execution of the cell only creates the definition of `say_hello`. The following indented line is part of the body of the function `say_hello`, but it is not executed yet.

[1]:
```
def say_hello():
    print('hello')
```

Note that although this function doesn't take any arguments, the parentheses `()` are still required in the header line.

We **call the function** with its name and parentheses `()`, as in the next cell.

[2]:
```
say_hello()
```
```
hello
```

When the function is called, Python executes the code inside the function. The function `say_hello` has only one line `print('hello')`, and the output of that line is shown after the cell.

If we forget the parentheses `()` when calling the function, Python will simply display the fact that `say_hello` is a function, as in the cell below.

[3]:
```
say_hello
```
[3]:
```
<function __main__.say_hello()>
```

Therefore, both a function definition and a function call must always include parentheses `()`, even if the function does not have any arguments.

To modify the function `say_hello` so that it accepts arguments, let's rewrite it to greet people with their names.

[4]:
```
def greeting(name):
    print(f'Hello {name}')
```

[5]:
```
greeting(name='John')
```
```
Hello John
```

The `return` Statement

The **return** statement is used with functions to pass data back to the caller. This is very convenient for reusing or saving the resulting variables from a function.

For example, the following function `add_two` increases the value of the input argument `x` by 2 and returns the result.

[6]:
```
def add_two(x):
    """Increase x by 2."""
    return x+2
```

[7]:
```
# Call the function
add_two(x=3)
```

[7]:
```
5
```

We can assign the returned value to a variable as in the next cell. Note the difference with the above cell: when the returned value is assigned to the variable `a`, the value is not displayed in the output of the cell. We will need to type it or use `print` to display the variable `a`.

[8]:
```
# Assign the result to the variable 'a'
a = add_two(x=5)
```

[9]:
```
# Show
a
```

[9]:
```
7
```

Also note in cell [8] that the first equal sign `a =` is used for assigning the value of the function to the variable name `a`, whereas the second equal sign in `x=5` is used for passing the value `5` to the argument `x` in the function.

Besides passing the outputs of a function, the `return` statement also immediately terminates the function and passes the execution control back to the caller. In general, the `return` statement doesn't need to be at the end of a function, and it can appear anywhere in a function body, and it can even appear multiple times. Consider the following example.

[10]:
```
def check_number(x):
    if x < 0:
        return
    if x > 100:
        return
    print(x)
```

[11]:
```
check_number(x=-3)
```

[12]:
```
check_number(x=125)
```

[13]:
```
check_number(x=50)
```

```
50
```

The first two calls to the function `check_number()` don't produce any output, because when the `return` statements are executed, they terminate the function immediately, before the `print` statement in the last line is reached.

This property of the `return` statements can be useful for **error checking** in a function. We can check several error conditions at the start of a function with return statements that terminate the code if there are any errors. If none of the error conditions are encountered, then the function can proceed with processing the block of statements.

```
def func():
    if error_condition1:
        return
    if error_condition2:
        return
    if error_condition3:
        return
    block of statements
```

A function can return multiple outputs, as in the following example.

[14]:
```
def multioutput(x):
    return x**2, x**3, x**4, x**5

multioutput(x=4)
```

[14]:
```
(16, 64, 256, 1024)
```

Also, a function can return any type of object in Python. For instance, the following function returns a list.

[15]:
```
def func1():
    """Return a list"""
    return ['foo', 'bar', 'baz', 'qux']
```

[16]:
```
# Call the function
func1()
```

[16]:
```
['foo', 'bar', 'baz', 'qux']
```

[17]:
```
# Or, assign the result to a variable 'b'
b = func1()
b
```

[17]:
```
['foo', 'bar', 'baz', 'qux']
```

[18]:
```
type(b)
```

[18]:
```
list
```

Or, a function can return a dictionary, as in this example.

[19]:
```
def build_person(first_name, last_name):
    """Return a dictionary of information about a person."""
    return {'first': first_name, 'last': last_name}

musician = build_person('jimi', 'hendrix')
musician
```

[19]:
```
{'first': 'jimi', 'last': 'hendrix'}
```

If a value is not provided after the `return` statement, the function returns the special Python data type `None`.

[20]:
```
def func2(x):
    x = x + 10
    return
```

[21]:
```
c = func2(x=5)
c
```

[22]:
```
type(c)
```

[22]:
```
NoneType
```

Similarly, if a function does not have a `return` statement, `None` data type will be returned. An example is presented in the next section.

Returning vs Printing Function Values

It is important to observe the difference between `print` and `return` statements in a function.

When the next function `add_two_print` is executed, the `print` statement displays the output `6`, however this value is not assigned to the variable `d` below, and instead the variable `d` has `None` data type. This means that the `print` statement can not be used to pass the output of a function to a variable.

[23]:
```
def add_two_print(x):
    print(x+2)
```

[24]:
```
# Call the function and assign it to variable 'd'
d = add_two_print(x=4)
```

```
6
```

[25]:
```
type(d)
```

[25]:
```
NoneType
```

Conversely, when the following function `add_two_return` is executed, and its value is assigned to the variable `e`, we can notice that the value of the variable `e` is 6 and its type is `integer`.

[26]:
```
def add_two_return(x):
    return x+2
```

[27]:
```
# Call the function and assign it to variable 'e'
e = add_two_return(x=4)
```

[28]:
```
e
```

[28]:
```
6
```

[29]:
```
type(e)
```

[29]:
```
int
```

Polymorphism

Note that Python does not require to specify the type of input objects to functions, and the same function can work with strings, numbers, or lists as input arguments. This behavior is called **polymorphism**, where polymorphism literally means occurrence in different forms. In other words, an operation depends on the type of objects being operated upon.

> **Polymorphism** in Python refers to using a single entity (e.g., operator, method) to represent different object types in different scenarios.

Let's see an example.

[30]:
```
def add_num(num1,num2):
    return num1+num2

# Call the function with numbers
add_num(4,5)
```

[30]:
```
9
```

[31]:
```
# Call the function with strings
add_num('one','two')
```

[31]:
```
'onetwo'
```

While the above function `add_num` performs addition for numbers, it does concatenation for strings. The example demonstrates polymorphism, since different operations are applied based on the type of input arguments to the function.

Conditional Statements in Functions

Functions in Python often contain conditional statements such as `if`, `else`, and `for` and `while` loops, to define relationships between the inputs and the outputs.

Let's see an example of a function to check if a number is prime (i.e., divisible only by 1 and itself).

[32]:
```
def is_prime(num):
    '''
    Naive method of checking for prime numbers.
    '''
    for n in range(2,num):
        if num % n == 0:
            print(num,'is not prime')
            break
    else: # If modulo is not zero, then it is prime
        print(num,'is prime!')
```

[33]:
```
is_prime(num=16)
```

```
16 is not prime
```

[34]:
```
is_prime(num=17)
```

```
17 is prime!
```

Note how the `else` statement is aligned under `for` and not `if`. This is because we want the `for` loop to exhaust all possibilities in the range before printing that the number is prime.

Also note how we `break` the code after the first print statement. As soon as we determine that a number is not prime we break out of the for loop.

Following is another example where for a given string the function returns a string in which every character from the original string is repeated three times.

[35]:
```
def threepeat(text):
    result = ''
    for char in text:
        result += char * 3
    return result
```

[36]:
```
# Check
threepeat('Mississippi')
```

[36]:
```
'MMMiiissssssiiissssssiiipppppppiii'
```

4.1.2 Argument Passing

As we mentioned, **arguments** are objects that are passed to functions as inputs. Arguments are also referred to as *input parameters* of a function.

Recall that functions do not need to have any arguments, as in the `say_hello` function above, and we used empty parentheses when we call such functions, as `say_hello()`.

Let's see an example where we used `message` and `character` as arguments to the function `print_box`.

[37]:
```
def print_box(message, character):
    print(character*10)
    print(message)
    print(character*10)
```

This function can be called in two ways by using positional arguments or keyword arguments.

Positional Arguments

Function call with positional arguments matches passed argument objects to the argument names in the function header by position, from left to right.

In the next cell we call the function `print_box`, and assign `Hi there!` to the argument `message` and `*` to the argument `character` for the function to display.

[38]:
```
print_box('Hi there!','*')
```

```
**********
Hi there!
**********
```

If we change the order of the arguments in the function call, the objects will be still passed based on their position.

[39]:
```
print_box('*', 'Hi there!')
```

```
Hi there!Hi there!Hi there!Hi there!Hi there!Hi there!Hi there!Hi there!Hi there!Hi␣
↪there!
*
Hi there!Hi there!Hi there!Hi there!Hi there!Hi there!Hi there!Hi there!Hi there!Hi␣
↪there!
```

Positional arguments are the recommended way for passing arguments with functions that take only one or two arguments. However, if a function takes many positional arguments it may be difficult to tell which argument is which.

Keyword Arguments

Function call with keyword arguments matches passed argument objects to the argument names in the function header by using the `name=value` syntax, where `name` is the keyword for the argument. Keyword arguments are recommended when a function needs to take many arguments, since each argument has a name and it is easy to see which argument object is matched to which argument name.

[40]:
```
print_box(message='Hi there!', character='*')
```

```
**********
Hi there!
**********
```

If keywords are not passed, then by default, arguments are matched by position, from left to right.

In this example, the integers 3, 5, and 2 are passed by position, where `a` is matched to 3, `b` is matched to 5, and `c` is matched to 2.

[41]:
```
def f(a, b, c):
    print(a, b, c)

f(3, 5, 2)
```

```
3 5 2
```

To call the same function with keyword arguments, we can use matching by name, and in this case, we don't need to worry about the order of the arguments when the function is called. In the next cell, the value `2` is passed to the name `c`, which is matched to the argument `c` in the function definition. The same holds for arguments `a` and `b`.

[42]:
```
f(c=2, a=3, b=5)
```

```
3 5 2
```

Keywords improve the code readability and make the function calls more self-documenting. For example, the function call `f(name='Bob', age=40, job='dev')` is much more meaningful than the function call `f('Bob', 40, 'dev')`, although they will both produce the same result.

However, passing keyword arguments that don't match any of the declared parameters in the function header results in an error.

[43]:
```
f(c=2, a=3, d=5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\1014038404.py in <module>
----> 1 f(c=2, a=3, d=5)
```

(continues on next page)

(continued from previous page)

```
TypeError: f() got an unexpected keyword argument 'd'
```

When a function is called with positional arguments, we must pass exactly as many argument objects as there are argument names in the function header. If we try to pass any number of arguments other than 3 to the above function, we will obtain an error (check the TypeError message below).

[ ]:
```
f(3, 5, 2, 1)
```

It is also possible to combine positional and keyword arguments in a single call. In the next call, first all positional arguments are matched based on their position going from left to right in the function header, and afterwards keywords arguments are matched by name.

[44]:
```
f(3, c=2, b=5)
```

```
3 5 2
```

Check the following error in the next cell: first 3 is assigned to `a`, afterward 2 is attempted to be assigned to `a` which results in an error.

[45]:
```
f(3, a=2, b=5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\1919709918.py in <module>
----> 1 f(3, a=2, b=5)

TypeError: f() got multiple values for argument 'a'
```

Also, when positional and keyword arguments are combined, positional arguments must be listed first, otherwise, we will get an error message.

[46]:
```
f(c=2, b=5, 3)
```

```
  File "C:\Users\Alex\AppData\Local\Temp\ipykernel_8724\3772222469.py", line 1
    f(c=2, b=5, 3)
                ^
SyntaxError: positional argument follows keyword argument
```

4.1.3 Default Values of Arguments

In the function definition header we can assign a **default value** to some arguments in the form `name=value`. Then, `value` becomes the default value for that argument. These arguments are also referred to as **optional arguments**. That is, if values are not passed to these arguments when the function is called, these arguments are assigned the default values.

For example, here is the same function that requires argument `a` and has default values for arguments `b` and `c`.

[47]:
```
# a is required argument, b and c are optional arguments
def f(a, b=5, c=2):
    print(a, b, c)
```

When we call this function, we must provide a value for argument `a` either by position or by keyword, and providing values for arguments `b` and `c` is optional. If we don't pass values to `b` and `c`, they default to 5 and 2, respectively.

[48]:
```
f(1)
```
```
1 5 2
```

[49]:
```
f(1, 2, 3)
```
```
1 2 3
```

[50]:
```
f(1, 4)
```
```
1 4 2
```

[51]:
```
f(1, c=6)
```
```
1 5 6
```

[52]:
```
f(c=6, a=6)
```
```
6 5 6
```

Note that there is a difference between the `name=value` syntax used in a function header and a function call. In the function header `name=value` specifies a **default** value for an optional argument. In the function call, `name=value` means a match-by-name **keyword** argument.

In both cases, `name=value` is a special syntax that is different from a regular assignment statement outside of a function (e.g., a Python line `a = 8` where the integer object `8` is assigned to the variable name `a`).

One small style detail that Python programmers do is to omit spaces around the = sign in the function header and call (e.g., `a=8`), to differentiate it from a general assignment statement `a = 8`.

4.1.4 Arbitrary Number of Arguments

Python also supports functions that take an arbitrary (i.e., variable) number of arguments. To achieve this, we need to use an asterisk sign * in front of the arguments name in the function definition header. Let's look at an example.

[53]:
```
def f1(*pargs):
    print(pargs)
    print(type(pargs))
```

When this function `f1` with ***pargs** is called, Python collects **positional arguments** as a tuple, and assigns the variable `pargs` to the tuple. The tuple can then be processed using regular tuple tools.

[54]:
```
f1(1,2)
```
```
(1, 2)
<class 'tuple'>
```

[55]:
```
f1(1, 2, 4, 10, 5, 100, 10000)
```
```
(1, 2, 4, 10, 5, 100, 10000)
<class 'tuple'>
```

Let's see an example where a function is used to calculate the average value of a collection of numbers entered by the user, where the user can enter as many numbers as they wish.

[56]:
```
def avg(*pargs):
    total = 0
    for i in pargs:
        total += i
    return total / len(pargs)
```

[57]:
```
# Call avg
avg(1, 2, 3)
```

[57]:
```
2.0
```

[58]:
```
# Call avg
avg(1, 2, 3, 4, 5, 6, 7, 8)
```

[58]:
```
4.5
```

In the above example, when the function `avg` is called, the passed arguments are packed into a tuple that the function uses with the name `pargs` to perform operations on the elements of the tuple.

Or, we could have even written the function `avg` in a more concise form.

[59]:
```
def avg(*pargs):
    return sum(pargs) / len(pargs)
```

[60]:
```
avg(2, 10, 100, 1000)
```

[60]:
```
278.0
```

As with other Python objects, we could have also used whatever variable name we wanted instead of `pargs` in the above examples. However, `pargs` or `args` are commonly used for this purpose, and if we use them, other people familiar with Python coding conventions will know immediately what we mean. Or, alternative the term `varargs` as an abbreviation for variable-length arguments is also used.

This feature of using a variable number of arguments is often referred to as `varargs`, after a variable-length argument tool in the C language.

Similarly, it is possible to use two asterisks signs ** in front of the arguments name in the function definition header. This case works only for **keyword arguments**. The arguments are collected into a dictionary, which can then be processed using regular dictionary tools.

[61]:
```
def f2(**kwargs):
    print(kwargs)
    print(type(kwargs))
```

[62]:
```
f2(a=1, b=2)
```

```
{'a': 1, 'b': 2}
<class 'dict'>
```

[63]:
```
f2(a=1, b=2, c=4, d=10, e=5, f=100, g=10000)
```

```
{'a': 1, 'b': 2, 'c': 4, 'd': 10, 'e': 5, 'f': 100, 'g': 10000}
<class 'dict'>
```

Let's next see one example of a function that concatenates a variable number of entered words by the user.

[64]:
```
def concatenate_words(**kwargs):
    result = ""
    for arg in kwargs.values():
        result = result + arg
    return result
```

[65]:
```
concatenate_words(item1='The', item2='quick', item3='brown', item4='fox')
```

[65]:
```
'Thequickbrownfox'
```

[66]:
```
concatenate_words(item1='The', item2='quick', item3='brown', item4='fox',
                  item5='jumped', items6='over')
```

[66]:
```
'Thequickbrownfoxjumpedover'
```

[67]:
```
concatenate_words(item1='The', item2='quick', item3='brown',
                  item4='fox', item5='jumped', items6='over',
                  item7='the', item9='lazy', item10='dog')
```

[67]:
```
'Thequickbrownfoxjumpedoverthelazydog'
```

When the function `concatenate_words` is called the passed keyword arguments are packed into a dictionary, which the function uses with the name `kwargs` to perform operations upon its elements. Notice in the code that the concatenated words are taken to be the values of the dictionary in `kwargs.values()`.

Function headers can also combine positional or keyword arguments, and arguments with preceding * and **. For instance, in the following function call, 1 is passed to `a` by position, 2 and 3 are collected into the `*pargs` (positional arguments) tuple, and `x` and `y` are collected in the `**kwargs` (keyword arguments) dictionary.

[68]:
```
def f3(a, *pargs, **kwargs):
    print(a, pargs, kwargs)
```

[69]:
```
f3(1, 2, 3, x=1, y=2)
```

```
1 (2, 3) {'x': 1, 'y': 2}
```

This type of argument definition in the function header is useful when we don't know the number of arguments that will be passed to a function when we write the code.

Unpacking Arguments in Function Calls

The * syntax can also be used in a function call. In this context, its meaning is opposite of its meaning in the function definition as it unpacks a collection of arguments, rather than builds a collection of arguments.

[70]:
```
def f4(a, b, c, d):
    print(a, b, c, d)
```

When the tuple `t1` is passed to the function `f4` with *, Python **unpacks** the tuple into the individual arguments `a`, `b`, `c`, and `d`.

[71]:
```
t1 = (2, 4, 6, 8)
f4(*t1)
```

```
2 4 6 8
```

In function calls, we don't even need to pass the arguments as a tuple with the * syntax, and instead we can pass a list, or any other *iterable object*.

[72]:
```
l1 = ['one', 'two', 'three', 'four']
f4(*l1)
```

```
one two three four
```

[73]:
```
s1 = 'spam'
f4(*s1)
```

```
s p a m
```

Similarly, the ** syntax in a function call unpacks a dictionary of key/value pairs into separate keyword arguments.

[74]:
```
d1 = {'a': 1, 'b': 3, 'c': 6, 'd': 9}
f4(**d1)
```

```
1 3 6 9
```

Note that if we tried to pass the tuple `t1` with the ** syntax, we will get an error message.

[75]:
```
f4(**t1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\621137479.py in <module>
----> 1 f4(**t1)

TypeError: __main__.f4() argument after ** must be a mapping, not tuple
```

To summarize again, the */** argument syntax in the function header collects a variable number of arguments, while in the function call it unpacks a variable number of arguments.

In both, one asterisk * applies to positional arguments, and two asterisks ** apply to keywords arguments.

Arguments Ordering Rules

The general syntax for function arguments is summarized in the table.

In a *function caller* (first four syntax rules in the table), normal values are matched by position. The `name=value` form is used to match keyword arguments by name. Using an `*iterable` or `**dict` in a function call allows to package up positional or keyword objects in iterables and dictionaries, respectively, and unpack them as separate, individual arguments when they are passed to the function.

In a *function header*, a simple name is matched by either position or name, depending on how the caller passed it. The `name=value` form specifies a default value. The `*name` form collects any extra unmatched positional arguments in a tuple, and the `**name` form collects extra keyword arguments in a dictionary. Any normal or defaulted argument names following a `*name` or an asterisk * are keyword-only arguments and must be passed by keyword in calls.

Syntax	Location	Interpretation
`func(value)`	Caller	Normal argument: matched by position
`func(name=value)`	Caller	Keyword argument: matched by name
`func(*iterable)`	Caller	Pass all objects in *`iterable`* as individual positional arguments
`func(**dict)`	Caller	Pass all key/value pairs in *`dict`* as individual keyword arguments
`def func(name)`	Function	Normal argument: matches any passed value by position or name
`def func(name=value)`	Function	Default argument value, if not passed in the call
`def func(*name)`	Function	Matches and collects remaining positional arguments in a tuple
`def func(**name)`	Function	Matches and collects remaining keyword arguments in a dictionary
`def func(*other, name)`	Function	Arguments that must be passed by keyword only in calls (3.X)
`def func(*, name=value)`	Function	Arguments that must be passed by keyword only in calls (3.X)

Figure source: Reference [1].

The steps that Python internally carries out to match arguments before assignment can roughly be described as follows:

1. Assign keyword arguments by position.
2. Assign keyword arguments by matching names.
3. Assign extra arguments to `*pargs` tuple.
4. Assign extra keyword arguments to `**kwargs` dictionary.
5. Assign default values to unassigned arguments in header.

After this, Python checks to make sure each argument is passed just one value; if not, an error is raised.

4.1.5 Namespace and Scope

It is very important to understand how Python handles the assigned variable names. In Python, the variable names are stored in a **namespace**.

> A **namespace** is a collection of names along with information about the object that each name references.

When an object is assigned to a name, Python internally creates a dictionary where the name is the key and the object is the value. This dictionary is updated with new keys and values as new names are created and objects are assigned. Python maintains several different namespaces, which include namespaces for each module, for each function, as well as a namespace for built-in Python functions.

Although there are different namespaces, Python may not be able to access each namespace from every part of the script. For instance, the names defined outside of a function cannot be accessed inside a function. The visibility of variable names to other parts in the code is determined by the scope.

> **Scope** is the portion of code from where a namespace can be accessed directly..

The benefit of having different namespaces in Python is that we can define and use variables within a function even if these variables have the same name as other variables defined in other functions or in the main program. In these cases, there will be no conflicts or interference between variables that have the same names, because they are kept in separate namespaces. This means that when we write code within a function we can use variable names and identifiers without worrying about whether they are already used elsewhere outside the function.

Let's see an example. Here, the name `x` is defined inside the function `printer`. This name `x` is not visible outside of the function. When we tried to access the name `x` in the cell outside the function `printer`, Python reported an error.

[76]:
```
def printer():
    x = 50   # x assigned in function: local name
    return x

x  # x accesed in module: global name
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\3041531381.py in <module>
      3     return x
      4
----> 5 x  # x accesed in module: global name

NameError: name 'x' is not defined
```

If we define the name `x` in the cell (i.e., in the module), then we can expect that we can access it without any errors.

[77]:
```
x = 25  # x assigned in module: global name

def printer():
    x = 50   # x assigned in function: local name
    return x

x
```

[77]:
```
25
```

The more important question is: what would be the output of `printer()`? Here we have two names `x`, where one is local (lives in the local namespace of the function `printer`) and the other one is global (lives in the global namespace of the cell).

Let's check.

[78]:
```
printer()
```

[78]:
```
50
```

Note that the value of `x` in the first cell is 25, and in the second is 50. Python follows a set of rules to decide which `x` variable is referenced in the code.

When Python searches for a name, it checks four scopes defined by the LEGB rule. The rule means that when a name is used, Python will first search the local (L) scope, then the scopes of any enclosing (E) functions, then the global (G) scope, and finally the built-in (B) scope. If the name is not found, Python reports an error.

LEGB Rule

L: Local scope — All names assigned within a `def` function are local (unless they are declared global in that function).

E: Enclosing function scope — Names in the local scope of any enclosing functions, from inner to outer functions.

G: Global (module) scope — Names assigned at the top level of a module file, or declared global within a `def` function with the `global` statement.

B: Built-in Python scope — Names preassigned in the built-in names module, such as `open`, `True`, `range`, etc.

Local Names

When we declare variables inside a function definition, they are visible only to the code inside the `def` function. Local variables are not related in any way to other variables having the same names used outside the function: i.e., the variable names are local to the function.

Examples:

[79]:
```
# x is local here:
def squared(x):
    return x**2  # x assigned in function: local name
```

[80]:
```
x = 50    # x assigned in module: global name

def func(x):
    print('x is', x)     # x is global
    x = 2      # x assigned in function: local name
    print('Changed local x to', x)   # x is local
```

[81]:
```
func(x)
```

```
x is 50
Changed local x to 2
```

[82]:
```
print('x is still', x)    # x is global
```

```
x is still 50
```

In the above example, the first time that we print the value of the name `x` with the first line in the function's body (i.e., the line `print('x is', x)`, Python uses the value of the variable declared in the main block, above the function definition. This means that variables defined in the main module are visible inside functions.

Next, we assign the value 2 to `x`. The name `x` is local to this function. So, when we change the value of `x` in the function, the variable `x` defined in the main block remains unaffected. That is, the local name is visible only within the function, and not visible outside the function.

With the last print statement `print('x is still', x)`, we display the value of `x` as defined in the main block, thereby confirming that it is actually unaffected by the local assignment within the previously called function.

Local variables serve as temporary names that are needed only while a function is running. They are removed from the memory after the function call exits.

In Jupyter, a quick way to test for global variables is to see if another cell recognizes the variable.

[83]:
```
print(x)
```

```
50
```

The `global` statement

If we want to assign a value to a name defined at the module level of the program (i.e., not inside any function), then we can use the `global` statement to indicate that the name is not local.

This allows to use the values of such variables outside the function. However, this is not encouraged and declaring global variables should be minimized, because it becomes unclear to the reader of the program as to where that variable's definition is.

Here is an example.

[84]:
```
x = 50    # x assigned in module: global name

def func():
    global x      # x is declared global
    print('This function is now using the global x!')
    print('Because of global, x is:', x)
    x = 2      # x assigned in function, but now it is global
    print('Now global x changed to', x)    # x is global
```

[85]:
```
func()
```

```
This function is now using the global x!
Because of global, x is: 50
Now global x changed to 2
```

[86]:
```
print('Value of x outside of func is:', x)  # x is global, assigned to 2 in the function
```

```
Value of x outside of func is: 2
```

The global statement is used to declare that `x` is a global variable - hence, when we assign a value to `x` inside the function, that change is reflected when we use the value of `x` in the main block.

We can specify more than one global variable using the same global statement, e.g. global x, y, z.

One last mention is that we can use the **globals()** and **locals()** functions to check what are our current local and global variables.

Built-in Function Names

Built-in names refer to standard names used in Python, like `open`, `True`, `None`, etc. Since Python searches last in the LEGB lookup for built-in names, this means that names in the local scope may override names in the built-in scope. For instance, if we create a local variable called `open`, then the built-in function for opening a file `open(data.txt')` will not work.

[87]:
```
def some_function():
    open = 'Hello'     # Local variable, hides the built-in open function
    open('data.txt')   # Error: this function no longer opens a file in this scope
    print(open)
```

[88]:
```
some_function()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\2867438862.py in <module>
```

(continues on next page)

(continued from previous page)

```
----> 1 some_function()

~\AppData\Local\Temp\ipykernel_8724\2724584223.py in some_function()
      1 def some_function():
      2     open = 'Hello'     # Local variable, hides the built-in open function
----> 3     open('data.txt')   # Error: this function no longer opens a file in this␣
↪scope
      4     print(open)

TypeError: 'str' object is not callable
```

Enclosing Function Names

Enclosing functions occurs when we have a function inside a function (that is, nested functions). With enclosing functions, Python first looks for a name in the local scope of a function, and next in the scopes of all enclosing functions, from inner to outer.

[89]:
```
x = 'This is a global name'   # x assigned in module: global name

def greet():
    # 'Greet' is an enclosing function to 'hello'
    x = 'Sammy'  # x assigned in enclosing function: enclosing function name
    def hello():
        print('Hello '+ x)
    hello()
```

[90]:
```
greet()    # x is defined in an enclosing function
```

```
Hello Sammy
```

Note that because the function `hello` is enclosed inside of the function `greet`, `hello` has access to the name `x` that is defined in the enclosing function. Also note that this did not change the value of the global name `x`.

[91]:
```
# Check if global
x
```

[91]:
```
'This is a global name'
```

4.1.6 The Importance of Python Functions

All programming languages support user-defined functions similar to Python, although in other languages functions may be referred to as subroutines, procedures, or subprograms.

The most important reason for using functions is **code abstraction and reusability**. For instance, when we write code we often need to implement tasks that are repeated in different locations in an application. Instead of replicating the code over and over again, we can just define a function that can be reused where needed. Also, if we need to modify the written code, it is easier to modify it only in one location in the defined function, instead of modifying all copies of the code scattered across many locations in the application.

Another reason for using functions is **code modularity**, since functions allow to break down complex processes into smaller blocks of code, organized into functions that focus on performing specific tasks. Such modularized code is more readable and understandable, and easier to maintain and update.

Docstrings

As we mentioned, the lines after the function header can be used to provide an optional description of the function, known as the **docstring** of the function. A docstring can include information about the purpose of the function, inputs arguments it takes, returned outputs values, or any other information that would be helpful to the users.

Here is an example of a docstring for the `avg` function that we used above.

[92]:
```
def avg(*args):
        """Returns the average of a list of numbers"""
        return sum(args) / len(args)
```

Docstrings are written inside quotes, and the recommended convention is to use three double-quote characters """, although any type of quotes are acceptable.

For more complex functions, multi-line docstrings are used, which typically consist of a summary line, followed by a blank line, followed by a more detailed description. The closing quotes should be on a separate line. Here is an example.

[93]:
```
def add_binary(a, b):
    '''
    Returns the sum of two decimal numbers in binary digits.

    Parameters:
            a (int): A decimal integer
            b (int): Another decimal integer
    Returns:
            binary_sum (str): Binary string of the sum of a and b
    '''
    binary_sum = bin(a+b)[2:]
    return binary_sum
```

We can type `help(function_name)` to display the docstring for any Python function.

[94]:
```
help(add_binary)
```

```
Help on function add_binary in module __main__:

add_binary(a, b)
    Returns the sum of two decimal numbers in binary digits.

    Parameters:
            a (int): A decimal integer
            b (int): Another decimal integer
    Returns:
            binary_sum (str): Binary string of the sum of a and b
```

Function Annotations

Function annotations in Python can be used to attach metadata to the input arguments and return values of a function. To add an annotation to a Python input argument, insert a colon `:` followed by an expression, and to add an annotation to the return value, add the characters `->` and an expression between the closing parenthesis of the argument list and the colon that terminates the function header.

Here is an example, where the annotations for the function indicate that the first argument is `int`, the second argument is `str`, and the return value is `float`.

[95]:
```
def func3(a: int, b: str) -> float:
        print(a, b)
        return(3.5)

func3(2, 'python')
```

```
2 python
```

[95]:
```
3.5
```

Annotations are optional, and they don't impose restrictions on the type of arguments passed to the function, nor have any impact on the execution of the code. They are simply metadata that provides information to the user regarding the arguments and the return values. Also, they can provide any other information about the arguments and return values, and not only the data type. Although it is possible to insert this information in the docstring of the function, placing it directly in the function definition adds clarity.

4.1.7 The lambda Expression

Similarly to using the `def` statement to define a new function, the `lambda` expression can also be used to create a new function in Python, but without assigning a name to the function. This is why `lambdas` are sometimes known as **anonymous (i.e., unnamed) functions**.

For example, let's use the `def` statement to create a simple function named `func1` that returns the sum of three numbers.

[96]:
```
def func1(x, y, z):
    return x + y + z

func1(2, 3, 4)
```

[96]:
```
9
```

The same can be achieved with a `lambda` expression. The newly created function can be assigned to a name `func2` which can be later called when needed.

[97]:
```
func2 = lambda x, y, z: x + y + z
func2(1, 2, 3)
```

[97]:
```
6
```

Let's compare the function objects `func1` and `func2` in the next cell. As we can see, the `lambda` expression creates a function object, which can be called later.

[98]:
```
func1
```

[98]:
```
<function __main__.func1(x, y, z)>
```

[99]:
```
func2
```

[99]:
```
<function __main__.<lambda>(x, y, z)>
```

The general syntax of a `lambda` expression consists of the keyword `lambda`, followed by one or more arguments, after the arguments comes a colon `:`, which is followed by an expression using the listed arguments.

```
lambda argument1, argument2,... argumentN : expression using arguments
```

Although `lambda` creates new functions in a similar way as `def`, there are a few differences.

- `lambda` is an expression, not a statement. Because of this, a `lambda` can appear in places a `def` is not allowed by Python's syntax, such as inside a list or inside a function call's arguments.
- `lambda`'s body is a single expression, not a block of statements. Because of that, `lambda` is less general than `def`.
- `lambda` is designed for coding simple functions, and `def` is designed for more complex tasks.

To illustrate the first bullet point above, consider the following list, which consists of three `lambda` functions defined inline within the list. A `def` won't work inside a list, because `def` is a statement, not an expression.

[100]:
```
# The items output a number raised to 2, 3, 4 power
list3 = [lambda x: x ** 2,
    lambda x: x ** 3,
    lambda x: x ** 4]

for x in list3:
    print(x(2))
```

```
4
8
16
```

The `lambda` expression allows to use default values for arguments, just like in `def` functions.

[101]:
```
# Default values for x, y, and z are defined in the lambda expression
func3 = lambda x=3, y=4, z=5: x + y + z
func3(1)
```

[101]:
```
10
```

Similarly, to introduce conditional logic in a `lambda` expression, we can use the `if-else` ternary expression that we mentioned in the earlier lecture on `if` tests (i.e., `a if x else b`).

For example, the following function returns the greater of two numbers.

[102]:
```
greater_number = (lambda x, y: x if x > y else y)
greater_number(5, 8)
```

[102]:
```
8
```

Although it is possible to encode quite complex logic expressions using `lambdas`, or even to nest `lambdas` within other `def` functions, `lambda` is still intended only to embed small pieces of code inline at the place it needs to be used. When there is a need for more complex logic, it is recommended to use `def` functions for simplicity and improved code readability.

7.4.2 4.2 Iterators

In the previous lecture we mentioned that iteration loops like `for` and `while` loops can work on any objects that are **sequences**, such as strings, lists, and tuples.

[103]:
```
# Iteration over items in a list
for x in ['book', 'pencil', 'board']:
    print(x, end = ' ') # print the items on the same line by adding an empty space ' ' ␣
↪after each
```

```
book pencil board
```

These sequences are *iterable objects* and loops iterate over their elements. For example, integer and floating-point numbers are not iterable objects, and if we try to iterate over them, we will get an error message.

[104]:
```
for x in 16:
    print(x)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\2559496794.py in <module>
----> 1 for x in 16:
      2     print(x)

TypeError: 'int' object is not iterable
```

Furthermore, `for` loops can work on other objects that are not sequences, such as files. The files can be scanned line by line, and therefore it is possible to iterate over them. All such objects in Python are called *iterable objects*.

> An object is considered an **iterable object** if it is either a sequence object, or an object that produces one result at a time when an iteration tool (like a `for` loop) is applied.

To explain how iteration over files works, let's create a simple file having two lines.

[105]:
```
# Creating a new file
myfile = open('data/test.txt','w')
myfile.write('This is a first line\n')
myfile.write('This is a second line\n')
myfile.close()
```

[106]:
```
# Reading the file line by line
myfile = open('data/test.txt','r')
myfile.readline()
```

[106]:
```
'This is a first line\n'
```

[107]:
```
myfile.readline()
```

[107]:
```
'This is a second line\n'
```

[108]:
```
myfile.readline()
```

[108]:
```
''
```

As we explained earlier, the iterator remembers the position in the file, and when the end-of-file is reached, the file cannot be read again without resetting the iterator to the beginning of the file.

[109]:
```
# If we try to read the file again, we cannot read the file since the iterator is at the␣
↪end of the file
myfile.readline()
```

[109]:
```
''
```

Files have an important method named `__next__()` that acts almost identically to `.readline()`. The only difference is that `__next__()` raises a StopIteration exception at the end-of-file (EOF).

[110]:
```
myfile = open('data/test.txt','r')
myfile.__next__()
```

[110]:
```
'This is a first line\n'
```

[111]:
```
myfile.__next__()
```

[111]:
```
'This is a second line\n'
```

[112]:
```
# StopIteration exception at the end-of-file
myfile.__next__()
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\334399950.py in <module>
      1 # StopIteration exception at the end-of-file
----> 2 myfile.__next__()

StopIteration:
```

[113]:
```
myfile.close()
```

And, in addition, there is also a Python built-in function `next()` that does the same thing as `__next__()`.

[114]:
```
myfile = open('data/test.txt','r')
next(myfile)
```

[114]:
```
'This is a first line\n'
```

[115]:
```
next(myfile)
```

[115]:
```
'This is a second line\n'
```

[116]:
```
# StopIteration exception at the end-of-file
next(myfile)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\2838511806.py in <module>
      1 # StopIteration exception at the end-of-file
----> 2 next(myfile)

StopIteration:
```

[117]:
```
myfile.close()
```

In Python, all iteration tools (like `for` loops) work internally by calling `__next__()` on each iteration and catching the StopIteration exception to determine when to exit.

For instance, the following cell shows code for reading the file using a `for` loop. Note also that using a `for` loop is preferred to using the `readline()` method shown above, because it is simpler to code, runs quicker, and is better in terms of memory usage (i.e., `readline()` loads the entire file into memory all at once, and it will not even work for large files that cannot fit into the memory space available on your computer).

[118]:
```
# Use file iterators to read by lines
for line in open('data/test.txt'):
    print(line, end = '')
```

```
This is a first line
This is a second line
```

Internally, in Python the above code is implemented somewhat similar to this next cell, where `__next__()` is called on each iteration to advance to the next position, and `try-except` is used to catch the StopIteration exception at the end-of-file.

[119]:
```
# Manual iteration
myfile = open('data/test.txt','r')
while True:
    try:
        line = myfile.__next__()
    except StopIteration:
        break
    print(line, end = '')
```

```
This is a first line
This is a second line
```

Not only the `for` loops, but all iteration tools work by calling `__next__()` on each iteration and catching the StopIteration exception. This includes list comprehensions, `in` membership tests, and other tools.

> An **iterator** is an object that uses a `__next__()` method to advance to the next item, and raises StopIteration at the end of the series of results.

In the above example `myfile` is an iterator, and it is also an iterable object.

In fact, all iterators are iterable objects, but not all iterable objects are iterators. For instance, although lists, strings, and tuples are iterable objects, we cannot directly use `__next__()` to iterate over them.

[120]:
```
L = [1, 2, 3]
# Raises an error message
L.__next__()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\3841945041.py in <module>
      1 L = [1, 2, 3]
      2 # Raises an error message
----> 3 L.__next__()

AttributeError: 'list' object has no attribute '__next__'
```

To obtain an iterator for a list, we need to use either the `__iter__()` method or the built-in function `iter()`.

[121]:
```
L = [1, 2, 3]
# Obtain an iterator
I = iter(L)
# Call iterator's next to advance to next item
I.__next__()
```

[121]:
```
1
```

[122]:
```
I.__next__()
```

[122]:
```
2
```

[123]:
```
I.__next__()
```

[123]:
```
3
```

[124]:
```
I.__next__()
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\3023204773.py in <module>
----> 1 I.__next__()

StopIteration:
```

We can perform a check to find out if an object is an iterator. As we learned, iterators for lists are different than the list object itself. This also holds for directories. For such objects, we must call `iter` to start iterating.

[125]:
```
iter(L) is L
```

[125]:
```
False
```

However, for files, the iterable object is also its own iterator, therefore it is not needed to use the `iter()` function to create an iterator.

[126]:
```
iter(myfile) is myfile
```

[126]:
```
True
```

The internal iteration for a list looks similar to this code.

[127]:
```
# Manual iteration
L = [1, 2, 3]
I = iter(L)
while True:
    try:
        X = I.__next__()
    except StopIteration:
        break
    print(X)
```

```
1
2
3
```

Many other functions in Python that scan objects from left to right perform iterations in a similar way. Examples are `sorted` for sorting items in an iterable; `zip` combines items from iterables; `enumerate` pairs items in an iterable with

relative positions; `filter` selects items for which a function is true, etc.

For example, note how `enumerate` works.

[128]:
```
for i, x in enumerate('hello'):
    print(i, x)
```

```
0 h
1 e
2 l
3 l
4 o
```

[129]:
```
# Iteration
S = enumerate('hello')
S.__next__()
```

[129]:
```
(0, 'h')
```

[130]:
```
S.__next__()
```

[130]:
```
(1, 'e')
```

To show all index-value pairs in `enumerate` we can wrap it in a list.

[131]:
```
list(enumerate('hello'))
```

[131]:
```
[(0, 'h'), (1, 'e'), (2, 'l'), (3, 'l'), (4, 'o')]
```

[132]:
```
# Compare to:
enumerate('hello')
```

[132]:
```
<enumerate at 0x1b85a99abc0>
```

The same holds for other functions.

[133]:
```
zip('abc', 'xyz')
```

[133]:
```
<zip at 0x1b85a9960c0>
```

[134]:
```
list(zip('abc', 'xyz'))
```

[134]:
```
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

[135]:
```
range(5)
```

[135]:
```
range(0, 5)
```

[136]:
```
list(range(5))
```

[136]:
```
[0, 1, 2, 3, 4]
```

7.4.3 4.3 Generators

Generator functions are a special type of functions that return one result at a time. This is different from the regular functions that we studied so far, which return all the values at the same time at the end of the execution, and after that they exit. Generators yield a value after value, by suspending and resuming the execution of the function from where they left off.

Generator functions are coded using the `def` statement similarly to regular functions; however, they use the `yield` statement at the end of the function block, instead of the `return` statement.

This allows generators to produce a series of values over time, rather than computing the returned values all at once and sending them back when the function is called.

> **Generators** are special functions that return an iterable set of items, one at a time. They are defined using a `def` statement, and they output values using a `yield` statement.

Here is one example of a simple generator function.

[137]:
```
def my_gen():
    n = 1
    print('This is printed first')
    yield n

    n = 2
    print('This is printed second')
    yield n

    n = 3
    print('This is printed at last')
    yield n
```

[138]:
```
for item in my_gen():
    print(item)
```

```
This is printed first
1
This is printed second
2
This is printed at last
3
```

Let's look at another example of the generator `gen_squares` which generates squared values of a series of numbers.

[139]:
```
# Generator function definition
def gen_squares(N):
    for i in range(N):
        # Generators use yield instead of return
        yield i ** 2
```

Because generators output a series of results, we can not call a generator in the same way as we would call a regular function. If we do, we will obtain a message that this object is a generator object.

[140]:
```
# The output is a generator object
gen_squares(5)
```

[140]:
```
<generator object gen_squares at 0x000001B85A99CC10>
```

We can call the generator `gen_squares` using a `for` loop, and the output in this case are the squares of the numbers 0, 1, 2, 3, and 4.

The generator `gen_squares` yields a value each time it is called within the `for` loop. When it is resumed, its prior state is restored, including the last values of its variables `i` and `N`, and the control picks up again immediately after the `yield` statement.

[141]:
```
for i in gen_squares(5):
    print(i)
```

```
0
1
4
9
16
```

Iterations with generator functions are similar to iterations over items in a list. The `__next__()` method starts the iteration and resumes it from where it last yielded a value, and raises a `StopIteration` exception when the end of the series of values is reached. Also note that for generators it is not required to apply the `iter()` method, since generators are their own iterators.

[142]:
```
y  = gen_squares(5)
y.__next__()
```

[142]:
```
0
```

[143]:
```
y.__next__()
```

[143]:
```
1
```

[144]:
```
y.__next__()
```

[144]:
```
4
```

[145]:
```
y.__next__()
```

[145]:
```
9
```

[146]:
```
y.__next__()
```

[146]:
```
16
```

[147]:
```
y.__next__()
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\81612537.py in <module>
----> 1 y.__next__()

StopIteration:
```

We can check if the iterator of the generator object (named `y` in this case) is the same as the generator, and as expected, the answer is yes.

[ ]:
```
iter(y) is y
```

Also, note that `yield` works only within the function block of statements, and if used outside of a function will get an error message.

[ ]:
```
yield 'hello'
```

An easy way to think about generators is to compare them to our computers. When we suspend a computer, it goes into a stand-by mode, and we can later continue using the computer and all of our programs are still there, just like they were when we left. Similarly, generators output some results, and they resume outputting the results the next time they are called.

By now, you may be asking yourself why and when we should use generators. They are used when the outputs of the functions are very large or when it takes a lot of computation to produce each output value. Generators are advantageous in terms of both memory use and performance in large programs.

One example application of generators is for machine learning applications with large datasets. For instance, latest datasets for image processing are often very large, and they consist of tens of thousands, or sometimes, millions of images. Loading such large datasets at once in the computer memory is not possible. Using generators allows to process such large datasets, by loading and processing a batch of images at a time that can fit into the computer's memory. Once a batch is processed, the generator will load the next batch and process it, and this step is repeated until all images are processed.

On the other hand, if you work with simpler programs and smaller datasets, using generator functions is not needed. The above example of generating squared values of a list of 5 numbers is too simple to illustrate the value of generators (because we could have used a `for` loop to achieve the same result); still, the example is useful for explaining the concept of generators.

Also, generators can yield items in any iterable object, like tuples, strings, dictionaries, and files.

Python also supports **generator expressions** which are very similar to list comprehensions, and support all the syntax that is used with list comprehensions. They are also called generator comprehensions. Differently from list comprehensions that use square brackets, generator expressions are enclosed in parentheses.

For the above example of squared values, the list comprehension has the following form.

[148]:
```
L = [x ** 2 for x in range(4)]
L
```

[148]:
```
[0, 1, 4, 9]
```

The corresponding generator expression is shown below.

[149]:
```
G = (x ** 2 for x in range(4))
G
```

[149]:
```
<generator object <genexpr> at 0x000001B85A9A5580>
```

Unlike list comprehensions, generator expressions do not build a list, but they return a generator object. This object is an iterator object, that yields one item at a time. The generator also retains the state of the variable `x` while it is active.

To display the outputs of the generator we can use a `for` loop, similar to generator functions.

[150]:
```
for item in G:
    print(item)
```

```
0
1
4
9
```

Or, if we would like to display all output results of the generator expression at once, we can simply wrap it in a list.

[151]:
```
list(x ** 2 for x in range(4))
```

[151]:
```
[0, 1, 4, 9]
```

Like generator functions, generator expressions provide memory-space optimization, since they do not require the entire result list to be constructed all at once, as the square-bracketed list comprehension does. Also like generator functions, they divide the work into smaller time slices as they yield results in piecemeal fashion, instead of making the caller wait for the full set to be created in a single call.

On the other hand, generator expressions may also run slightly slower than list comprehensions in practice, so they are best used only for very large result sets, or applications that cannot wait for full results generation.

7.4.4 Appendix: Additional Functions Info

The material in the Appendix is not required for quizzes and assignments.

Function Decorators

A **decorator** is a design pattern in Python that allows to add new functionality to an existing object without modifying its structure. Decorators are typically called before the definition of a function that is to be decorated by using the syntax `@decorator`.

Using decorators is also called *metaprogramming* because a part of the program tries to modify another part of the program at execution time.

Let's consider the following simple function `divide`, which accepts two arguments `a` and `b`. We know it will give an error if we pass in `b` as 0.

[152]:
```
def divide(a, b):
    return a/b
```

[153]:
```
divide(12,3)
```

[153]:
```
4.0
```

[154]:
```
divide(2,0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\2734703358.py in <module>
----> 1 divide(2,0)

~\AppData\Local\Temp\ipykernel_8724\560253271.py in divide(a, b)
      1 def divide(a, b):
----> 2     return a/b

ZeroDivisionError: division by zero
```

Let's assume that we would like to modify the function `divide` so that it checks for the case of division by 0 that will cause the error.

One way to achieve this is to create a new function that will take as an argument the function `divide` and modify it. This way, we can apply the same functionality to other similar functions as well, if we needed to.

The following function named `smart_divide` does exactly that.

[155]:
```
def smart_divide(func):
    def inner_function(a, b):
        print("Divide", a, "and", b)
        if b == 0:
            print("Cannot divide by 0")
            return
        return func(a, b)
    return inner_function
```

Now, we can use the function `divide` as an argument in the new function `smart_divide`. We can name the new function `my_divide`.

[156]:
```
my_divide = smart_divide(divide)
```

[157]:
```
my_divide(12,3)
```

```
Divide 12 and 3
```

[157]:
```
4.0
```

[158]:
```
my_divide(12,0)
```

```
Divide 12 and 0
Cannot divide by 0
```

Modifying existing functions using a decorator is a common construct in Python, and the syntax uses the `@` symbol along with the name of the decorator function placed above the definition of the function to be decorated.

[159]:
```
@smart_divide
def divide(a, b):
    print(a/b)
```

[160]:
```
divide(12,3)
```

```
Divide 12 and 3
4.0
```

[161]:
```
divide(12,0)
```

```
Divide 12 and 0
Cannot divide by 0
```

Using `@smartdivide` in the above code is equivalent to writing `divide = smart_divide(divide)`.

The new functionality added by the decorator function `smart_divide` to the original function `divide` can be seen as similar to packing a gift, where the decorator acts as a wrapper. The actual gift inside the wrapper does not alter, but now it looks pretty since it got decorated.

Or, in other words, the decorator is a function that modifies another function. When the decorated function is invoked through its original name `divide`, the decorator is applied to augment the original function in some way.

Example 2

One more simple example follows, where the function `ordinary()` does not take any input arguments.

[162]:
```
def ordinary():
    print("Ordinary function")

ordinary()
```

```
Ordinary function
```

A decorator function called `make_pretty()` wraps around the code of the argument function `func_1` and inserts additional print statements before and after the function `func_1`.

[163]:
```
def make_pretty(func_1):    # the input argument to the decorator is another function,␣
↪here called 'func_1'
    def inner():            # this is the wrapper for the function
        print("The function will be decorated")
        func_1()            # this executes the initial function
        print("The function was decorated")
    return inner
```

Let's apply the decorator, and call the function `ordinary()` after it has been decorated.

[164]:
```
@make_pretty
def ordinary():
    print("Ordinary function")

ordinary()
```

```
The function will be decorated
Ordinary function
The function was decorated
```

In practice, we don't even need to ever use decorators, and we can achieve the same results by just using `make_pretty(ordinary)` as shown below. However, decorators have advantages over such approach, and are commonly used.

[165]:
```
def ordinary():
    print("Ordinary function")

ordinary = make_pretty(ordinary)

ordinary()
```

```
The function will be decorated
Ordinary function
The function was decorated
```

Functions as Arguments, Inner Functions, and Returns in Other Functions

In order to better understand decorators, we will now take a step back and explain several related concepts in Python.

As we mentioned before, everything in Python is an **object**. Names that we assign to objects are simply identifiers bound to these objects. Functions are no exceptions, and they are objects too.

Functions can be passed as arguments to another function. Here is an example, where the functions `increase` and `decrease` are passed as arguments to the function `operate`. Functions like `operate` that take other functions as arguments are also called **higher order functions**.

[166]:
```
def increase(x):
    # Increse x by 1
    return x + 1

def decrease(x):
    # Decrease x by 1
    return x - 1

def operate(func, x):   # Operate takes 2 arguments: a function `func` and a number `x`
    result = func(x)
    return result
```

[167]:
```
# call 'operate' with 'increase' as argument
operate(increase,3)
```

[167]:
```
4
```

[168]:
```
# call 'operate' with 'decrease' as argument
operate(decrease,3)
```

[168]:
```
2
```

Furthermore, other functions can be nested under another function, and are referred to as **inner functions**.

Here's an example of a function with two inner functions.

[169]:
```
def parent():
    print("Printing from the parent() function")

    def first_child():
        print("Printing from the first_child() function")

    def second_child():
        print("Printing from the second_child() function")

    second_child()
    first_child()
```

[170]:
```
parent()
```

```
Printing from the parent() function
Printing from the second_child() function
Printing from the first_child() function
```

Note that the order in which the inner functions are defined does not matter, and what matter is the order in which the inner functions are called within the `parent()` function. That is, `second_child` is called first, and afterward `first_child` is called.

As we know, the inner functions are in the local scope to the outer (enclosing) function `parent()`, and they only exist inside the `parent()` function as local variables. If we try calling `first_child()` we will get an error.

[171]:
```
first_child()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\2276402280.py in <module>
----> 1 first_child()

NameError: name 'first_child' is not defined
```

Python also allows to use other functions as **returns** of a function. The following example returns one of the inner functions from the outer `parent()` function, based on an `if` test.

[172]:
```
def parent(num):
    def first_child():
        return "Hi, I am Emma"

    def second_child():
        return "Call me Liam"

    if num == 1:
        return first_child
    else:
        return second_child
```

[173]:
```
my_func1 = parent(1)
my_func1()
```

[173]:
```
'Hi, I am Emma'
```

[174]:
```
my_func2 = parent(5)
my_func2()
```

[174]:
```
'Call me Liam'
```

[175]:
```
type(my_func2)
```

[175]:
```
function
```

[176]:
```
# my_func_2 is a function object
my_func2
```

[176]:
```
<function __main__.parent.<locals>.second_child()>
```

Also note that in the previous example we executed the inner functions within the parent function, because we used `first_child()` and `second_child()`. However, in this last example, we did not add parentheses to the inner functions `first_child` and `second_child` in the return lines. That way, we obtained a reference to each function that we can later call, that is `my_func1` and `my_func2` above are function objects, and we need to call them with parentheses (e.g., `my_func1()`) in order to obtain the print statement. If we don't use parentheses (as in `my_func2`), Python will just display that this is a function object.

Compare to the code below, where we called the inner functions in the return lines.

[177]:
```
def parent(num):
    def first_child():
        return "Hi, I am Emma"

    def second_child():
        return "Call me Liam"

    if num == 1:
        return first_child()
    else:
        return second_child()
```

[178]:
```
my_func2 = parent(5)
my_func2
```

[178]:
```
'Call me Liam'
```

[179]:
```
type(my_func2)
```

[179]:
```
str
```

[180]:
```
# my_func2 here is a string and not a function, we cannnot call it
my_func2()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\1447515405.py in <module>
      1 # my_func2 here is a string and not a function, we cannnot call it
----> 2 my_func2()

TypeError: 'str' object is not callable
```

One more similar simple example is shown below where a nested function is returned in an outer function. Each time we call `first_function()`, `nested_function()` is returned.

[181]:
```
def first_function():  # created 1st function

    print('First line in the called function')

    def nested_function():  # Created 2nd function (nested)
        print('Hello, this is inside the nested function')
    return nested_function
```

[182]:
```
new_function = first_function()

new_function()
```

```
First line in the called function
Hello, this is inside the nested function
```

[183]:
```
# Compare to the following
first_function()
```

```
First line in the called function
```

[183]:
```
<function __main__.first_function.<locals>.nested_function()>
```

General Syntax of a Decorator

The following code provides a general syntax of a decorator. The `decorator_function` takes a function called `func` as its argument, and returns a modified version of it called `new_function`.

```
def decorator_function(func):                    # the 'decorator_function' will be␣
↪later invoked using the @ syntax
    def new_function(*args, **kwargs):           # this function is often named 'wrapper
↪' or 'inner' function
        # Perform actions with `func`, `args` and `kwargs`
        ...

    return new_function

@decorator_function
def my_function(arg1, arg2, arg3, ....):         # my_function is passed as an argument␣
↪to 'decorator_function'
    # Peform some actions, equivalent to 'my_function = decorator_function(my_function)'


my_function(arg1=v1, arg2=v2, arg3=v3,....)     # call 'my_function', with the values v1,
↪ v2, v3, ... passed to 'new_function'
```

Why Use Decorators

Like other advanced Python tools, decorators are never strictly required from a purely technical perspective: we can implement the same functionality using simple helper functions or other techniques. Or, we can always manually modify the code in a function instead of using a decorator.

However, imagine the scenario where instead of adding some functionality to one function, we have a large package with hundreds of functions to which we would like to add the same functionality. To do that, we would need to copy the same thing over and over again, which is error-prone, as we could miss one place where it is required, or paste it in the wrong place. Using decorators allows to modify all functions in a manner that is less error-prone.

There are also other reasons for using decorators that go beyond just avoiding repetitive typing. When we need to modify the logic in our programs, we can modify the logic in just one place, instead of trying to find related code everywhere, and perhaps make mistakes along the way.

Furthermore, decorators have a very explicit syntax with the @ symbol, which makes them easier to spot than helper function calls that may be arbitrarily far-removed from the functions upon which they act.

In summary, decorators offer advantages in terms of both code maintenance and consistency. Although the choice to use decorators is still subjective, their advantages are compelling enough so that they are adopted by many Python users.

Keyword-Only Function Arguments

As we explained, the order of arguments in a function header is as follows, where first come normal arguments (`args`), then default arguments (`defargs`, such as `name=value`), followed by * positional arguments, and last are ** keyword arguments.

```
def some_function(args, defargs, *pargs, **kwargs):
```

Python also allows to specify arguments that must be passed by keyword only. Such arguments should be placed in the function header after `*pargs` in the list of arguments.

In the following example, `a` may be passed by name or position, `b` collects any extra positional arguments, and `c` must be passed by **keyword only**. I.e., `c` cannot be passed as a positional argument.

[184]:
```
def keyword_only(a, *b, c):
    print(a, b, c)
```

[185]:
```
keyword_only(1, 2, c=3)
```

```
1 (2,) 3
```

If `c` is passed by position, we will get an error message.

[186]:
```
keyword_only(1, 2, 3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\1433115857.py in <module>
----> 1 keyword_only(1, 2, 3)

TypeError: keyword_only() missing 1 required keyword-only argument: 'c'
```

We can also use an asterisk character * by itself in the arguments' list to indicate that a function does not accept a variable-length argument list, but still expects all arguments after the * to be passed as keywords.

[187]:
```
def keyword_only2(a, *, b, c, d=4):
    print(a, b, c, d)
```

[188]:
```
keyword_only2(1, c=5, b=3, d=7)
```

```
1 3 5 7
```

[189]:
```
keyword_only2( d=7, c=5, a=2, b=3)
```

```
2 3 5 7
```

[190]:
```
# d uses the default value from the function header d=4, d is optional in the function␣
↪call
keyword_only2(6, c=5, b=3)
```

```
6 3 5 4
```

[191]:
```
# Keyword only arguments that don't have a default value are required, not optional
keyword_only2(6)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\2458061151.py in <module>
      1 # Keyword only arguments that don't have a default value are required, not␣
↪optional
----> 2 keyword_only2(6)

TypeError: keyword_only2() missing 2 required keyword-only arguments: 'b' and 'c'
```

Note also that keyword-only arguments cannot be specified after `**kwargs`.

[192]:
```
def keyword_only3(a, **b, c):
    print(a, b, c)
```

```
  File "C:\Users\Alex\AppData\Local\Temp\ipykernel_8724\3712509756.py", line 1
    def keyword_only3(a, **b, c):
                           ^
SyntaxError: invalid syntax
```

Also, ** cannot appear by itself in the function header, unlike * that can appear by itself.

[193]:
```
def keyword_only4(a, **, b, c):
    print(a, b, c)
```

```
  File "C:\Users\Alex\AppData\Local\Temp\ipykernel_8724\55769500.py", line 1
    def keyword_only4(a, **, b, c):
                          ^
SyntaxError: invalid syntax
```

Keyword-only arguments must appear before `**kwargs` in the function header.

[194]:
```
def keyword_only5(a, *b, c, **d):
    print(a, b, c, d)
```

[195]:
```
keyword_only5(1, 2, 3, c=4, d=5)
```

```
1 (2, 3) 4 {'d': 5}
```

Similarly, when keyword-only arguments are passed in a function call, they must appear before `**kwargs` form.

[196]:
```
keyword_only5(1, *(2, 3, 4), c=5, **{'d':6, 'e':7})
```

```
1 (2, 3, 4) 5 {'d': 6, 'e': 7}
```

Keyword-only arguments are helpful when we want to make sure that positional arguments are not incorrectly matched in a function call. For instance, if we write a function to move a file from one folder to another folder, and we have an argument `overwrite` (=True or False), we may want to specify that `overwrite` is always called by keyword, to make sure that `overwrite` is not accidentally matched to another argument, causing to inadvertently delete another file.

Positional-only Arguments

As of Python 3.8, function parameters can also be declared as positional-only, by following them in the list of arguments with a forward slash `/`. In the next example, the arguments `x` and `y` must be passed as positional arguments and can not be passed as keyword arguments.

[197]:
```
def func5(x, y, /, z):
    print(x, y, z)

func5(1, 2, 3)
```

```
1 2 3
```

If `x` and `y` are passed as keyword arguments, that will raise an error.

[198]:
```
func5(x=1, y=2, z=3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8724\2185976999.py in <module>
----> 1 func5(x=1, y=2, z=3)

TypeError: func5() got some positional-only arguments passed as keyword arguments: 'x, y'
```

More on Argument Passing in Python

In Python, arguments are passed by automatically assigning objects to local variable names. All arguments are passed by references to objects stored in the memory of the local computer. This means that the objects are assigned to arguments, and these objects are only referenced, i.e., copies of the objects are not created. This way, it is possible to pass objects to functions in our programs without making multiple copies of the objects along the way.

When *mutable* objects (like lists or dictionaries) are passed to arguments in a function, the mutable objects can be changed in-place, and the results may affect the called objects. On the other hand, *immutable* objects (like integers, floats, tuples) cannot be changed when passed as arguments.

The following example demonstrates this.

[199]:
```
def func1(a, b):
    a = 2
    b[0] = 'spam'
```

[200]:
```
# X is a global name, it is assigned to integer, which is an immutable object
X = 1
# L is a global name, it is assigned to a list, which is a mutable object
L = [1, 2]

# Pass both immutable and mutable objects as arguments to the function
func1(X, L)

# Check how they are affected: X is unchanged, L is different!
X, L
```

[200]:
```
(1, ['spam', 2])
```

In the example, argument `a` is a local variable name in the function's scope, and when `X` is called, since `X` is an integer (immutable object), `a` will just reference to its global name, which points to the object that has a value `1`.

Argument `b` is also a local variable name, but it is passed to the list `L`, which is a mutable object. This results in an in-place object change, and the assignment to `b[0]` in the function impacts the value of `L`.

If we don't want in-place changes within functions to impact mutable objects we pass to them, we can simply pass copies of mutable objects as function arguments.

[201]:
```
L = [1, 2]

# Instead of L we pass a copy of L
func1(X, L.copy())
# L didn't change this time
L
```

[201]:
```
[1, 2]
```

7.4.5 References

1. Mark Lutz, "Learning Python," 5-th edition, O-Reilly, 2013. ISBN: 978-1-449-35573-9.
2. Pierian Data Inc., "Complete Python 3 Bootcamp," codes available at: https://github.com/Pierian-Data/Complete-Python-3-Bootcamp.
3. Joh Sturtz at The Real Python, "Defining Your Own Python Function," available at: https://realpython.com/defining-your-own-python-function/#functions-in-python.
4. Python - Made with ML, Goku Mohandas, codes available at: https://madewithml.com/.
5. Eric Matthes, "Python Crash Course," No Starch Press, 2016, ISBN-13: 978-1-59327-603-4.
6. Primer on Python Decorators, available at: https://realpython.com/primer-on-python-decorators/.
7. Python Decorators at Programiz, available at: https://www.programiz.com/python-programming/decorator.
8. Pierian Data Inc., "Complete Python 3 Bootcamp," codes available at: https://github.com/Pierian-Data/Complete-Python-3-Bootcamp.

BACK TO TOP

7.5 Lecture 5 - Object-Oriented Programming

7.5.1 5.1 Overview

Object-oriented programming (OOP) is a programming approach to structuring programs based on the concept of **objects**, which can contain *data* and *behavior* in the form of *attributes* and *methods*, respectively. For instance, an object could represent a person with attributes like name, age, and address, and methods such as walking, talking, and running. Or, it could represent an email with attributes like a recipient list, subject, and body, and methods like adding attachments and sending.

In OOP, computer programs are designed by defining objects that interact with one another. In Python, the main tool for achieving OOP are Python **classes**. Classes are created using the `class` statement. Then, from classes we can construct object **instances**, which are specific objects created from a particular class.

Besides OOP, other programming paradigms on which other languages are based include procedural, functional, and logic programming paradigms.

7.5.2 5.2 Defining a Class

The `class` Statement

Let's define a class `Dog` by using the `class` statement and the name of the class. It is a convention in Python to begin class names with an uppercase letter, and module and function names with a lowercase letter. This is not a requirement, but if you follow this naming convention it will be appreciated by others who are to use your codes.

[1]:
```
# Create a new class called Dog
class Dog:
    """Class object for a dog."""
    pass
```

[2]:
```
# Create an instance object of the class Dog
sam = Dog()

print(type(sam))
```

```
<class '__main__.Dog'>
```

In the above code, inside the class definition we currently have just the `pass` command, which is only a placeholder for the code that we intend to write afterwards and means do nothing for now.

Classes can be thought of as blueprints for creating objects. When we defined the class `Dog` using the line `class Dog:`, we didn't actually create an object.

To create an object of the class `Dog`, we called the class by its name and a pair of parentheses (and optionally we can pass arguments in the parentheses as we did with functions in Python). That is, we **instantiated** the Dog class, and `sam` is now the reference to our new instance of the Dog class. Or, the action of creating instances (objects) from an existing class is known as **instantiation**.

The name `sam` is referred to as a **class instance**, or **instance object**, or just an **instance**. We will use these terms interchangeably.

We can create many instances of the class by calling the class with `Dog()`. For example, below we created two Dog instances `sam` and `frank`. Note that although they are both instances of the class Dog, they represent two distinct objects.

[3]:
```
sam = Dog()
frank = Dog()
```

(continues on next page)

(continued from previous page)

```
sam == frank
```

[3]:
```
False
```

Note below that the two instances `sam` and `frank` have different memory addresses, shown after `at` in the cell outputs. The addresses for these two instances in your computer's memory will be different than those shown here.

[4]:
```
sam
```

[4]:
```
<__main__.Dog at 0x2c132164160>
```

[5]:
```
frank
```

[5]:
```
<__main__.Dog at 0x2c132164550>
```

In summary:

> **Classes** serve as instance factories. They provide attributes and methods that are inherited by all the instances created from them.
>
> **Instances** represent the concrete objects of a class. Their attributes consist of information that varies per specific object. Their methods describe behavior that is different for specific objects.

Class objects can have *attributes* and *methods*.

> An **attribute** is an individual characteristic of an instance of the class. Or, attributes are variables that hold class data.
>
> A **method** is an operation that is performed with the instance of the class. Or, methods are functions that provide behavior to class objects.

5.2.1 Attributes

As we explained, attributes allow us to attach data to class objects. Python classes can have two types of attributes: instance attributes and class attributes.

Instance Attributes

In Python, **instance attributes** are defined by using the `__init__()` constructor method. The term *init* is abbreviated from *initialize*, since it is used to initialize the attributes of class instances.

In the parentheses of the `__init__()` method first `self` is listed, and afterwards the attributes are listed.

The syntax for creating attributes is:

```
def __init__(self, attribute1, attribute2, ...):
    self.attribute1 = attribute1
    self.attribute2 = attribute2
```

For example:

[6]:
```
class Dog:
    def __init__(self, breed, name):
        self.breed = breed
        self.name = name
```

[7]:
```
# Create an instance from the 'Dog' class by passing breed and name, assign it to the␣
↪name 'sam'
sam = Dog(breed='Labrador', name='Sam')
# Create another instance from the 'Dog' class by passing breed and name, assign it to␣
↪the name 'frank'
frank = Dog(breed='Huskie', name='Frank')
# Create another instance from the 'Dog' class by passing breed and name, assign it to␣
↪the name 'my_dog'
my_dog = Dog(breed='Terrier', name='Scooby')
```

The `__init__()` method is present in almost every class, and it is used to initialize newly created class instances by passing attributes. In the above example, the attributes *breed* and *name* are the arguments to the special method `__init__()`. Each attribute in a class definition begins with a reference to the class instance, which by convention is named `self`, such as in `self.breed = breed`.

When we created the instances of the class Dog, `__init__()` initialized these objects by passing the assigned values for *breed* and *name* to the instances *sam*, *frank*, and *my_dog*. In the `__init__()` method, the word `self` is the newly created instance. Therefore, for the instance *sam*, the line `self.breed = breed` is equivalent to stating `sam.breed = 'Labrador'`. Similarly, `self.name = name` is equivalent to stating `sam.name = 'Sam'`. Similarly, for the instance *frank*, `self.breed = breed` is equivalent to stating `frank.breed = 'Huskie'` and `self` refers to the instance *frank*.

Notice again that `sam`, `frank`, and `my_dog` are three separate instances of the `Dog` class, and they have their own attributes, i.e., different breed and name.

Accessing Instance Attributes

The syntax for accessing an attribute of a class instance uses the **dot operator**.

```
instance.attribute
```

We can therefore access the attributes `breed` and `name` as in the next examples.

[8]:
```
# Access the 'breed' attribute of the class instance 'sam'
sam.breed
```

[8]:
```
'Labrador'
```

[9]:
```
# Access the 'breed' attribute of the class instance 'frank'
frank.breed
```

[9]:
```
'Huskie'
```

[10]:
```
# Access the 'name' attribute of the class instance 'my_dog'
my_dog.name
```

[10]:
```
'Scooby'
```

Note that we cannot access the instance attributes through the class, as in `class.attribute`, since they are specific to concrete instances of the class. If we try to do that, we will get an Attribute Error.

[11]:
```
Dog.name
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_32936\4053790346.py in <module>
----> 1 Dog.name

AttributeError: type object 'Dog' has no attribute 'name'
```

In general, it is possible to create new classes without the `__init__()` construction method. This is shown below, where we used a `def` statement to introduce a method called `enterinfo` to the class `Dog`.

[12]:
```
class Dog:
    def enterinfo(self, breed):
        self.breed = breed
```

[13]:
```
sam = Dog()
sam.enterinfo(breed='Labrador')
```

[14]:
```
sam.breed
```

[14]:
```
'Labrador'
```

However, in this case, we need to first create a new class instance `sam` as shown above, and afterward assign the `breed` attribute using the `enterinfo()` method.

On the other hand, by using the `__init__()` method, we can initialize the instance attributes at the same time when the new instance is created. Therefore, using `__init__()` is preferred and always recommended. Without `__init__()`, an empty instance is created, and we need to initialize it afterwards.

In addition, we can dynamically attach new instance attributes to existing class objects that we have already created. In the next cell, the new attribute `age` is attached to the instance `sam`. However, it is preferred to define instance attributes inside the class definition, since it makes the code more organized and makes it easier for others to understand or debug our code.

[15]:
```
sam.age = 3
print(sam.age)
```

```
3
```

Modifying Instance Attributes

We can modify the attributes of an instance by using the dot `.` notation and an assignment statement, as in:

```
instance.attribute = new_value
```

[16]:
```
frank.name
```

[16]:
```
'Frank'
```

[17]:
```
# Modify attribute
frank.name = 'Franki'
frank.name
```

[17]:
```
'Franki'
```

To delete any instance attribute, use the `del` keyword.

[18]:
```
del frank.name
```

[19]:
```
# Error, the name attribute does not exist for 'frank'
print(frank.name, frank.breed)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_32936\2831644747.py in <module>
      1 # Error, the name attribute does not exist for 'frank'
----> 2 print(frank.name, frank.breed)

AttributeError: 'Dog' object has no attribute 'name'
```

The above code does not delete the attribute `name` for the other class instances.

[20]:
```
# The name attribute still exist for 'my_dog'
my_dog.name
```

[20]:
```
'Scooby'
```

Class Attributes

Class attributes in Python are also referred to as *class object attributes*. The class attributes are the same for all instances of the class.

For example, we could create the attribute *species* for the Dog class, as shown in the next cell. Regardless of their breed, name, or other attributes, all dog instances will have the attribute `species = 'mammal'`. The instances of the class Dog in the next cell also have the *instance attributes* `breed` and `name` which can be unique for each class instance.

We apply this logic in the following manner.

[21]:
```
class Dog:

    # Class attribute
    species = 'mammal'

    # Instance attributes
    def __init__(self, breed, name):
        self.breed = breed
        self.name = name
```

[22]:
```
# Create an instance from the 'Dog' class by passing breed and name
sam = Dog('Labrador','Sam')
```

Accessing class attributes is the same as accessing instance attributes.

[23]:
```
# Access class attributes
sam.species
```

[23]:
```
'mammal'
```

[24]:
```
# Access instance attributes
sam.name
```

[24]:
```
'Sam'
```

Note that the class attribute `species` is defined directly in the body of the `class` definition, outside of any methods in the class. Also by convention, the class attributes are placed before the `__init__()` method.

Also, we can access class attributes through the class via `class.attribute`, as in the next example.

[25]:
```
Dog.species
```

[25]:
```
'mammal'
```

Modifying Class Attributes

We cannot modify class attributes via assignment to class instances. In the next example, we used an assignment statement `frank.species = 'bird'` to modify the attribute `species` of the class instance `frank` to `bird`.

[26]:
```
frank = Dog(breed='Huskie', name='Frank')
```

[27]:
```
frank.species
```

[27]:
```
'mammal'
```

[28]:
```
# Reassing the attribute 'species' to 'bird'
frank.species = 'bird'
```

This didn't change the class attribute for the newly created class instance `my_dog`, as shown below. Instead, the reassignment `frank.species = 'bird'` created a new instance attribute `species` for the class instance `frank` that has the same name as the class attribute `species`.

[29]:
```
my_dog = Dog(breed='Terrier', name='Scooby')
```

[30]:
```
# The class attribute of the new instance is still 'mammal'
my_dog.species
```

[30]:
```
'mammal'
```

We can change the class atribute via assignment when using the class name, as shown in the next example.

[31]:
```
Dog.species = 'animal'
```

[33]:
```
sam = Dog('Labrador','Sam')
```

[34]:
```
sam.species
```

[34]:
```
'animal'
```

In summary:

- Class attributes are defined in the body of the `class` definition directly. Class attributes are common to the class. Their data is the same for all instances of the class.
- Instance attributes are defined inside the `__init__()` method within the `class` definition. Instance attributes belong to a concrete instance of the class. Their data is specific to that concrete instance of the class.

The __dict__ Attribute

Both classes and instances in Python have a special attribute called `__dict__`. This attribute is a dictionary, with the keys being the attribute names and the values are the attached attribute values. For a class instance `__dict__` holds the instance attributes, and for a class `__dict__` holds class attributes and methods.

[35]:
```
my_dog.__dict__
```

[35]:
```
{'breed': 'Terrier', 'name': 'Scooby'}
```

[36]:
```
frank.__dict__
```

[36]:
```
{'breed': 'Huskie', 'name': 'Frank', 'species': 'bird'}
```

[37]:
```
Dog.__dict__
```

[37]:
```
mappingproxy({'__module__': '__main__',
              'species': 'animal',
              '__init__': <function __main__.Dog.__init__(self, breed, name)>,
              '__dict__': <attribute '__dict__' of 'Dog' objects>,
              '__weakref__': <attribute '__weakref__' of 'Dog' objects>,
              '__doc__': None})
```

Python also allows to change the value of existing instance attributes through `__dict__`, or even to add new attributes through `__dict__`.

5.2.2 Methods

Methods are functions defined inside the body of a class. By defining it inside the class, we establish a relationship between the method and the class. Because methods are functions, they can take arguments and return values.

In a Python class, we can define three different types of methods:

- Instance methods, which take the current instance `self` as their first argument.
- Class methods, which take the current class `cls` as their first argument.
- Static methods, which take neither the class nor the instance.

This section describes instance methods, as the most common type of methods in classes. Class methods and static methods are described in the Appendix section.

Instance Methods

Instance methods are functions defined inside the body of a class, designed to perform operations on the class objects.

Methods have access to all attributes for an instance of the class. They can access and modify the attributes through the argument `self`.

We can basically think of methods as regular functions, with one major difference that the first argument of the method is always the instance object referenced through `self`.

Technically, even the word `self` is a convention, and any other term can be used instead of `self`. However, if you use another word, that would be very unusual for other coders using your code.

Let's see an example of creating a class for a `Circle` shown below. The objects of this class have three methods: `getArea` which calculates the area, `getCircumference` which calculates the circumference, and `SetRadius` which allows to change the attribute `radius`.

[38]:
```
class Circle:
    pi = 3.14

    # Circle gets instantiated with a radius (default is 1)
    def __init__(self, radius=1):
        self.radius = radius

    # Method for getting Area
    def getArea(self):
        return self.radius * self.radius * self.pi

    # Method for getting Circumference
    def getCircumference(self):
        return self.radius * self.pi * 2

    # Method for resetting Radius
    def setRadius(self, new_radius):
        self.radius = new_radius
```

Notice that within the methods `getArea` and `getCircumference` we used the notation `self.radius` to reference the instance attribute `radius` which we defined with the `__init__` method inside the body of the class.

Similarly, we defined the class attribute `pi = 3.14` inside the body of the class, and since we can access it as an attribute of new instances, we used the notation `self.pi` to reference it within the methods `getArea` and `getCircumference`. As we explained in the previous section, we can also access class attributes through the class name, that is, we could have used the notation `Circle.pi` within the methods `getArea` and `getCircumference` to get access to the class attribute `pi`.

The two methods `getArea` and `getCircumference` don't take any other arguments except `self`. The method `setRadius` takes another argument `new_radius`, and it allows to change the value of the current attribute `radius`.

The methods are accessed by using the dot notation.

```
instance.method()
```

[39]:
```
# Let's call it
c = Circle()

print('Radius is: ', c.radius)
print('Area is: ', c.getArea())
print('Circumference is: ', c.getCircumference())
```

```
Radius is:  1
Area is:  3.14
Circumference is:  6.28
```

Now let's change the radius with the method `setRadius` and see how that affects the `Circle` object:

[40]:
```
c.setRadius(3)

print('Radius is: ', c.radius)
print('Area is: ', c.getArea())
print('Circumference is: ', c.getCircumference())
```

```
Radius is:  3
Area is:  28.26
Circumference is:  18.84
```

Notice again in the above cell that when we call `getArea` and `getCircumference` methods, we don't need to provide a value for the `self` argument. Python takes care of that step, and it automatically passes the class instance to `self`.

However, if we wish, we can manually provide the desired class instance when calling these methods. To do this though, we need to call the method on the class, as shown next.

[41]:
```
Circle.getCircumference(c)
```

[41]:
```
18.84
```

If we try to call the methods on the instance `c`, that will raise an exception.

[42]:
```
c.getCircumference(c)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_32936\521085191.py in <module>
----> 1 c.getCircumference(c)

TypeError: getCircumference() takes 1 positional argument but 2 were given
```

Shown next is one more example of a class `Customer` with attributes `name` and `balance`, and methods `withdraw` and `deposit`.

[43]:
```
class Customer():
    """A customer of ABC Bank with a checking account. Customers have the
    following properties:

    Attributes:
        name: A string representing the customer's name.
        balance: A float tracking the current balance of the customer's account.
    """

    def __init__(self, name, balance=0.0):
        """Return a Customer object whose name is *name* and starting
        balance is *balance*."""
        self.name = name
        self.balance = balance

    def withdraw(self, amount):
        """Return the balance remaining after withdrawing *amount*
        dollars."""
        if amount > self.balance:
            raise RuntimeError('Amount greater than available balance.')
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        """Return the balance remaining after depositing *amount*
        dollars."""
```

(continues on next page)

(continued from previous page)

```
        self.balance += amount
        return self.balance
```

[44]:
```
# Create a new instance
bob = Customer('Bob Smith', 1000)
```

[45]:
```
bob.withdraw(100)
```

[45]:
```
900
```

[46]:
```
bob.deposit(400)
```

[46]:
```
1300
```

[47]:
```
# Based on the exception in the 'withdraw' method
bob.withdraw(1600)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_32936\323204834.py in <module>
      1 # Based on the exception in the 'withdraw' method
----> 2 bob.withdraw(1600)

~\AppData\Local\Temp\ipykernel_32936\2930014863.py in withdraw(self, amount)
     18         dollars."""
     19         if amount > self.balance:
---> 20             raise RuntimeError('Amount greater than available balance.')
     21         self.balance -= amount
     22         return self.balance

RuntimeError: Amount greater than available balance.
```

Polymorphism in Classes

We learned about polymorphism in functions, and we saw that when functions take in different arguments, the actions depend on the type of objects. Similarly, in Python **polymorphism** exists with classes, where different classes can share the same method name, and these methods can perform different actions based on the object they act upon.

Let's see an example.

[48]:
```
class Dog:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Woof!'

class Cat:
    def __init__(self, name):
        self.name = name

    def speak(self):
```

(continues on next page)

(continued from previous page)

```
        return self.name+' says Meow!'

niko = Dog('Niko')
felix = Cat('Felix')

print(niko.speak())
print(felix.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Here we have a `Dog` class and a `Cat` class, and each has a `speak()` method. When called, each object's `speak()` method returns a result that is unique to the class of the instance. This demonstrated polymorphism, because we passed in different object types to the `speak()` method, and we obtained object-specific results from the same method.

Naming Conventions in Classes

The recommended naming convention for Python classes is to use capitalized names, and for longer names each word is capitalized and connected without underscores. For example, examples of classes in the machine learning library `Keras` include `Conv2DTranspose`, `CheckpointCallback`, `BatchNormalization`, etc.

Another naming convention is to include a leading underscore in the names of attributes and methods (e.g., `_radius`, `_calculate_area()` in the class `Circle`) to communicate them as **non-public** attributes and methods. All regular names (such as `radius` and `calculate_area()`) are **public** attributes and methods.

Public members are intended to be part of the official interface or API of the classes, while non-public members are not intended to be part of the API. This naming convention indicates that the non-public members should not be used outside their defining class. However, the naming convention does not prevent direct access. Non-public members exist only to support the internal implementation of a given class and may be removed at any time, so we should not rely on them.

7.5.3 5.3 Inheritance

For our programs to be truly object-oriented, it is required that they use inheritance hierarchy. **Inheritance** is the process of creating a new class by reusing the attributes and methods from an existing class. This way, we can edit only what we need to modify in the new class, and this will override the behavior of the old class.

The newly formed inheriting class is known as a **subclass** or **child class** or **derived class**, and the class it inherits from is known as a **superclass** or **parent class** or **base class**.

Important benefits of inheritance are code reuse and reduction of complexity of a program, because the child classes override or extend the functionality of parent classes.

Let's see an example by incorporating inheritance. In this example, we have four classes: `Animal`, `Dog`, `Cat`, and `Fish`. The `Animal` is the parent class (superclass), and `Dog`, `Cat`, and `Fish` are the child classes (subclasses).

Note that when defining the child classes `Dog`, `Cat`, and `Fish`, the parent class `Animal` is listed in parentheses in the class header, i.e., `class Dog(Animal)`.

[49]:
```
class Animal:
    def __init__(self):
        print("Animal created")

    def whoAmI(self):
```

(continues on next page)

(continued from previous page)

```
        print("Animal")

    def eat(self):
        print("Eating")


class Dog(Animal): # The class Dog inherits the functionalities of the class Animal
    def __init__(self):
        Animal.__init__(self)   # This can also be replaced with: super().__init__()
        print("Dog created")

    def whoAmI(self):
        print("Dog")

    def bark(self):
        print("Woof!")


class Cat(Animal): # The class Cat inherits the functionalities of the class Animal
    def __init__(self):
        # The line Animal.__init__(self) is missing in the Cat class
        print("Cat created")

    def whoAmI(self):
        print("Cat")


class Fish(Animal): # The class Fish inherits the functionalities of the class Animal
    # attributes are not specified

    def whoAmI(self):
        print("Fish")
```

Let's create instances of the child classes.

[50]:
```
d = Dog()
```

```
Animal created
Dog created
```

[51]:
```
# Note the difference in the attributes in comparison to Dog
c = Cat()
```

```
Cat created
```

[52]:
```
# Note the difference in the attributes in comparison to Cat
f = Fish()
```

```
Animal created
```

Parent classes typically provide generic and common functionality that we can reuse throughout multiple child classes. In this sense, the `Animal` class provides properties that are common for most other animals.

Child classes **inherit attributes and methods** from the parent class. For instance, notice that if call the `eat()` method

with the class instances of `Dog` and `Fish`, the word `Eating` is printed. Although the method `eat()` is not defined in the classes `Dog` and `Fish`, the instances inherit the method from the parent class `Animal`.

[53]:
```
d.eat()
```
```
Eating
```

[54]:
```
f.eat()
```
```
Eating
```

The child classes not only inherit attributes and methods from the parent class, but they can also **modify attributes and methods** existing in the parent class. This is shown by the method `whoAmI()`. When this method is called, Python searches for the name first in the child class, and if it is not found, afterwards it searches in the parent class. In this case, `whoAmI()` method is found in the child classes `Dog`, `Cat`, and `Fish`.

[55]:
```
d.whoAmI()
```
```
Dog
```

[56]:
```
c.whoAmI()
```
```
Cat
```

Finally, the child class `Dog` **extends the functionality of the parent class** by defining a new `bark()` method that does not exist in the `Animal` class.

[57]:
```
d.bark()
```
```
Woof!
```

Similar to inheritance in nature, only child classes inherit from the parent class, and the parent class does not inherit attributes and methods from the child classes.

One more example follows, where `Person` is a parent class, and `Manager` is a child class of `Person` and inherits attributes and methods.

[58]:
```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        self.pay = int(self.pay * (1 + percent + bonus))
```

[59]:
```
# Create a new instance of Person
bob = Person('Bob Smith', pay=50000)
bob.giveRaise(percent=0.1)  # 50000 * (1+ 0.1) = 50000 * 1.1 = 55000
bob.pay
```
[59]:
```
55000
```

[60]:
```
# Create a new instance of Manager
tom = Manager('Tom Jones', 'mgr', 50000)
print(tom.name, tom.job, tom.pay)
```

```
Tom Jones mgr 50000
```

[61]:
```
# On a salary of 50,000, giveRaise for Person applied 10% raise, and giveRaise for␣
↪Manager applied 10% bonus
tom.giveRaise(percent=0.1, bonus=0.1)   # 50000 * (1+ 0.1 + 0.1) = 50000 * 1.2 = 60000
tom.pay
```

[61]:
```
60000
```

Another way to define the method `giveRaise` for the `Manager` child class is by using the syntax below `superclass.method(self, arguments)`, as in `Person.giveRaise(self, percent + bonus)` shown below. Note that this is different from the syntax above `self.method(arguments)` as in `self.pay = int(self.pay * (1 + percent + bonus))`. However, this coding approach using `self.method(arguments)` does not rely on the superclass `Person`, and if `Person` is changed, the code will not work as expected. Therefore, it is preferred to use the syntax `superclass.method(self, arguments)`.

[62]:
```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)
```

[63]:
```
tom = Manager('Tom Jones', 'mgr', 50000)
# On a salary of 50,000, giveRaise for Person applied 10% raise, and giveRaise for␣
↪Manager applied 10% bonus
tom.giveRaise(percent=0.1, bonus=0.1)   # Person.giveRaise(.10+0.10) = Person.
↪giveRaise(0.20)   # 50000 * 1.2 = 60000
tom.pay
```

[63]:
```
60000
```

The super() function

The `super()` function in Python returns a temporary object of the parent class that then allows to call methods from the parent class in child classes. This allows to define new methods in the child class with minimal code changes.

For instance, in the example below, a parent class `Rectangle` is defined, and a child class `Cube` is created that inherits from `Rectangle`. To calculate the volume of a Cube, the child class `Cube` inherited the method `area()` from the class `Rectangle` via `super().area()`. Since the method `volume()` for a cube relies on calculating the area of a single face, rather than reimplementing the area calculation, we use the function `super()` to extend the area calculation. The function `super()` returns an object of the superclass, and allows to call the method `volume()` directly through `super().area()`.

[64]:
```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width
```

(continues on next page)

(continued from previous page)

```
    def perimeter(self):
        return 2 * self.length + 2 * self.width

class Cube(Rectangle):
    def __init__(self, length, width, height):
        self.length = length
        self.width = width
        self.height = height

    def volume(self):
        face_area = super().area()
        return face_area * self.height
```

[65]:
```
cube1 = Cube(4, 4, 2)
cube1.volume()
```

[65]:
```
32
```

One more example is provided next, with a parent class `Person` and child class `Student`. Note that in the definition of the `Student` class, we called the `__init__()` function from the superclass to initialize the attributes `student_name`, `student_age`, and `student_residence`. The call to the parent class `super().__init__(student_name, student_age, student_residence)` is equivalent to calling the function as `Person.__init__(self, student_name, student_age, student_residence)`.

[66]:
```
class Person:

    def __init__(self, name, age, residence):
        self.name = name
        self.age = age
        self.residence = residence

    def show_name(self):
        print(self.name)

    def show_age(self):
        print(self.age)

class Student(Person):

    def __init__(self, student_name, student_age, student_residence, student_id):
        super().__init__(student_name, student_age, student_residence)
        self.studentId = student_id

    def show_id(self):
        print(self.studentId)
```

[67]:
```
# Create an object of the child class
student1 = Student("Max", 22, "Moscow", "100022")
student1.show_name()
```

```
Max
```

[68]:
```
student1.show_id()
```
```
100022
```

Class Hierarchies

Using inheritance, we can build class hierarchies, also known as inheritance trees. A **class hierarchy** is a set of closely related classes that are connected through inheritance and arranged in a tree-like structure. The class or classes at the top of the hierarchy are the parent classes, while the classes below are derived classes or child classes.

Therefore, classes at the top of the hierarchy are generic classes with common functionality, while classes down the hierarchy are more specialized and they inherit attributes and methods from their parent classes and also have their own attributes and methods.

Let's revisit again the example with the animals, where the following tree hierarchy will be created.

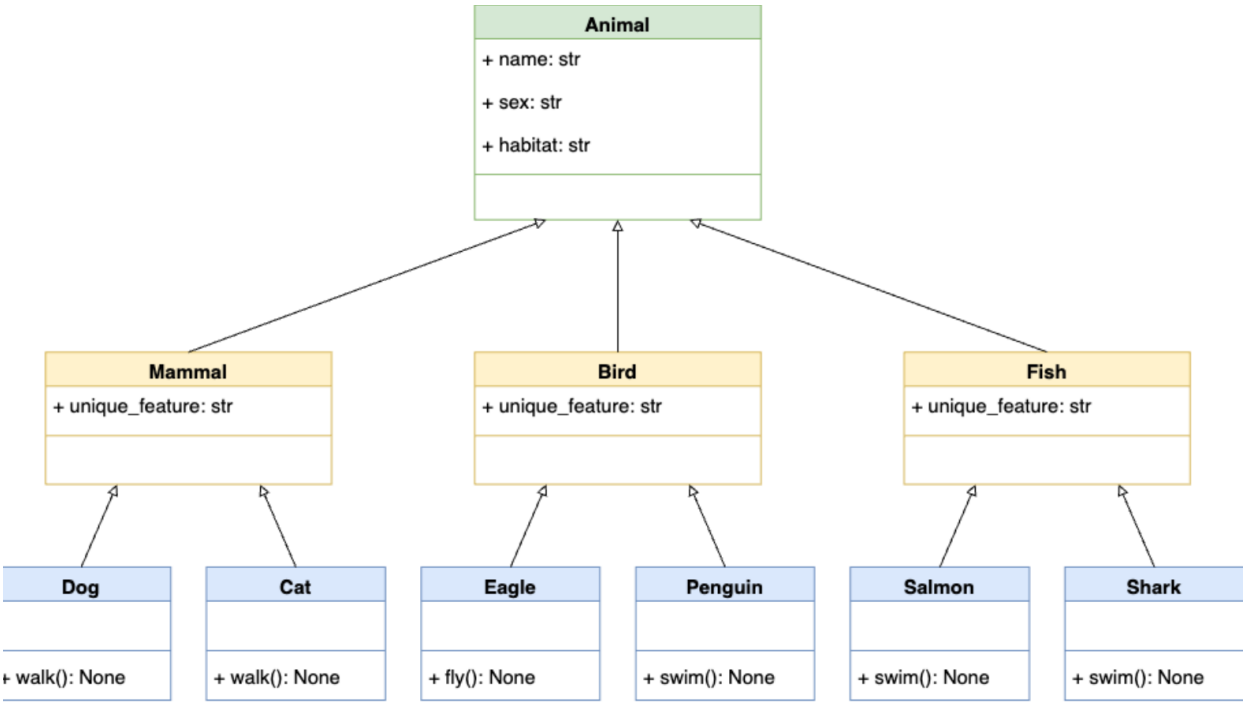


Figure source: Reference [3].

In this hierarchy, a parent class `Animal` is at the top. Below this class, we have subclasses like `Mammal`, `Bird`, and `Fish`, which inherit the attributes and methods from the class `Animal`. At the bottom level, we can have classes like `Dog`, `Cat`, `Eagle`, `Penguin`, `Salmon`, and `Shark`. E.g., `Dog` and `Cat` are both mammals and animals, and they inherit from both of these superclasses and have their own attributes and methods.

[69]:
```
class Animal:
    def __init__(self, name, sex, habitat):
        self.name = name
        self.sex = sex
        self.habitat = habitat

class Mammal(Animal):
    unique_feature = "Mammary glands"
```

(continues on next page)

(continued from previous page)

```
class Bird(Animal):
    unique_feature = "Feathers"

class Fish(Animal):
    unique_feature = "Gills"

class Dog(Mammal):
    def walk(self):
        print("The dog is walking")

class Cat(Mammal):
    def walk(self):
        print("The cat is walking")

class Eagle(Bird):
    def fly(self):
        print("The eagle is flying")

class Penguin(Bird):
    def swim(self):
        print("The penguin is swimming")

class Salmon(Fish):
    def swim(self):
        print("The salmon is swimming")

class Shark(Fish):
    def swim(self):
        print("The shark is swimming")
```

[70]:
```
d = Dog('Fido', 'M', 'Europe')
```

[71]:
```
d.unique_feature
```

[71]:
```
'Mammary glands'
```

[72]:
```
d.walk()
```

```
The dog is walking
```

7.5.4 5.4 Special Methods

Python has many other built-in methods which can be used with user-defined classes. These methods are also known as **special methods** or **magic methods**. Similar to the `__init__()` method, all special methods have leading and trailing double underscores (also called *dunders*).

For instance, the list of special methods for mathematical operators in Python involve the following.

```
a + b       a.__add__(b)
a - b       a.__sub__(b)
a * b       a.__mul__(b)
```

(continues on next page)

(continued from previous page)

```
a / b       a.__truediv__(b)
a // b      a.__floordiv__(b)
a % b       a.__mod__(b)
a << b      a.__lshift__(b)
a >> b      a.__rshift__(b)
a & b       a.__and__(b)
a | b       a.__or__(b)
a ^ b       a.__xor__(b)
a ** b      a.__pow__(b)
-a          a.__neg__()
~a          a.__invert__()
abs(a)      a.__abs__()
```

Special methods for item access in sequences involve the following.

```
len(x)      x.__len__()
x[a]        x.__getitem__(a)
x[a] = v    x.__setitem__(a,v)
del x[a]    x.__delitem__(a)
```

Other type of special methods are used for access to object attributes. These include:

```
x.a         x.__getattr__(a)
x.a = v     x.__setattr__(a,v)
del x.a     x.__delattr__(a)
```

And there are other types of special methods that are not listed above.

The use of special methods with user-defined classes is also called **operator overloading** in OOP, because these methods allow the new instances of our user-defined classes to exhibit the behaviors of the applied special methods. For example, the operator + is implemented using the special method `__add__()` and it can perform addition of numbers, concatenation of strings, etc. Operator overloading in Python is an example of *polymorphism* in Python. Note that the term polymorphism is more general, and it describes actions performed upon different objects in a different way based on the object, as we saw in the example from the previous section.

By implementing special methods into our user-defined classes, our classes can behave like built-in Python types.

Sequence Length with `__len__()`

We can implement the special method `__len__()` in our custom classes, which will allow us to use `len()` with the instances of the class.

In the example below, we used the method `__len__()` in the class `Employee`, which returns the length of the attribute `self.pay`.

[73]:
```
class Employee:

    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    def __len__(self):
        return len(self.pay)
```

[74]:
```
bob = Employee(name='Bob Smith', pay=[50000, 55000, 53000, 60000])
print(bob.name, bob.pay)
```

```
Bob Smith [50000, 55000, 53000, 60000]
```

[75]:
```
# Length of the bob.pay object
len(bob.pay)
```

[75]:
```
4
```

[76]:
```
sue = Employee(name='Sue Jones', pay=[50000, 60000])
print(sue.name, sue.pay)
```

```
Sue Jones [50000, 60000]
```

[77]:
```
len(sue.pay)
```

[77]:
```
2
```

See the Appendix for additional information about special methods for classes in Python.

7.5.5 5.5 When to Use Classes

Classes allow to leverage the power of Python while writing and organizing code. The benefits of using classes include:

- Reuse code and avoid repetition: we can define hierarchies of related classes, where the parent classes at the top of a hierarchy provide common functionality that we can reuse later in the child classes down the hierarchy. This allows to reuse code and reduces code duplication.
- Group related data and behaviors in a single entity: classes allow to group together related attributes and methods in a single entity. This helps you better organize code using modular entities that can be reused across multiple projects.
- Abstract away the implementation details of concepts and objects: classes allow to abstract away the implementation details of core concepts and objects. This helps provide the users with intuitive interfaces to process complex data and behaviors.

In conclusion, Python classes can help write more organized, structured, maintainable, reusable, flexible, and user-friendly code. appears, then go for it.

On the other hand, we should not use classes for everything in Python, since in some situations, they can overcomplicate our solutions. Sometimes, writing a couple of functions are enough for solving a problem.

For example, we don't need to use classes when we need to:

- Store only data: if there are no any methods inside the body of a class, we can use a dictionary or a named tuple instead.
- Provide a single method: if a class has only one method, it would be better to use a function instead.
- When a fucntionality is available through built-in types or third-party classes: in that case, we should avoid creating custom classes.

Also, there are other situations where we may not need to use classes, such as: in short and simple programs with simple logic and data structures, in performance-critical programs where classes may slow down the performance, when working in a team with a coding style that doesn't rely on classes, etc.

Therefore, although classes provide many benefits, they don't need to be used in every situations. Often, it is preferred to begin with a simple but working code, and if there is a need to use classes, then go for it.

7.5.6 Appendix: Additional OOP Info

The material in the Appendix is not required for quizzes and assignments.

Static Methods and Class Methods

In Section 5.2 above we studied **instance methods**, and we explained that they are applied to class instances through the use of the keyword `self`.

In Python there are also **static methods** and **class methods**, that are defined inside a class and are not connected to a particular instance of that class. These methods are created with the built-in decorators `@staticmethod` and `@classmethod`.

The following code shows the difference in the syntax between instance method, `@classmethod` and `@staticmethod`.

[78]:
```
class MyClass:
    def instance_method(self, arg1, arg2, argN):
        return 'instance method called', self

    @classmethod
    def classmethod(cls, arg1, arg2, argN):
        return 'class method called', cls

    @staticmethod
    def staticmethod(arg1, arg2, argN):
        return 'static method called'
```

Static Methods with `@staticmethod`

In Python and other programming languages, a **static method** is a method that does not require the creation of an instance of a class. For Python, it means that the first argument of a static method is not `self`, but a regular positional or keyword argument. Also, a static method can have no arguments at all, as in the following example.

In general, static methods are used to create helper functions that have a logical connection with the class but do not have access to the attributes or methods of the class, or to the class instances.

[79]:
```
class Cellphone:
    def __init__(self, brand, number):
        self.brand = brand
        self.number = number

    def get_number(self):
        return self.number

    @staticmethod
    def get_emergency_number():
        return "911"
```

[80]:
```
Cellphone.get_emergency_number()
```

[80]:
```
'911'
```

In this example, `get_number()` is a regular instance method of the class and requires the creation of an instance. The method `get_emergency_number()` is a static method because it is decorated with the `@staticmethod` decorator.

Also note that `get_emergency_number()` does not have `self` as the first argument, which means that it does not require the creation of an instance of the `Cellphone` class.

Again, `get_emergency_number()` can just work as a standalone function, and it does not need to be defined as a static method. However, it makes sense and is intuitive to put it in the `Cellphone` class because a cellphone should be able to provide the emergency number.

Here is one more example of using a static method. The method `is_full_name()` just checks whether the entered name for a student consists of more than one string.

[81]:
```
class Student():

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @staticmethod
    def is_full_name(name_str):
        names = name_str.split(' ')
        return len(names) > 1
```

[82]:
```
scott = Student('Scott',  'Robinson')
```

[83]:
```
# call the static method
Student.is_full_name('Scott Robinson')
```

[83]:
```
True
```

[84]:
```
# call the static method
Student.is_full_name('Scott')
```

[84]:
```
False
```

And one more example of using `@staticmethod` follows. In order to convert the slash-dates to dash-dates, we used the function `toDashDate` within the `Dates` class. It is a static method because it doesn't need to access any properties of the class `Dates` through `self`. It is also possible to create a function `toDashDate()` outside the class, but since it works for dates, it is logical to keep it inside the `Dates` class.

[85]:
```
class Dates:
    def __init__(self, date):
        self.date = date

    def getDate(self):
        return self.date

    @staticmethod
    def toDashDate(slash_date):
        return slash_date.replace("/", "-")
```

[86]:
```
date1 = Dates("15/12/2016")
date1.getDate()
```

[86]:
```
'15/12/2016'
```

[87]:
```
date2 = Dates.toDashDate("15/12/2016")
date2
```

[87]:
```
'15-12-2016'
```

In addition, static methods are used when we don't want subclasses of a superclass to change or override a specific implementation of a method. Because `@staticmethod` is ignorant of the class it is attached to, we can use it in subclasses just as it was defined in the superclass.

In the following code, `DatesWithSlashes` is derived from the superclass `Dates`. We wouldn't want the subclass `DatesWithSlashes` to override the static method `toDashDate()` because it only has a single use, i.e., change slash-dates to dash-dates. Therefore, we will use the static method to our advantage by overriding `getDate()` method in the subclass so that it works well with the `DatesWithSlashes` class.

[88]:
```
class Dates:
    def __init__(self, date):
        self.date = date

    def getDate(self):
        return self.date

    @staticmethod
    def toDashDate(date):
        return date.replace("/", "-")

class DatesWithSlashes(Dates):
    def getDate(self):
        return Dates.toDashDate(self.date)
```

[89]:
```
date1 = Dates("15/12/2016")
date1.getDate()
```

[89]:
```
'15/12/2016'
```

[90]:
```
date2 = DatesWithSlashes("15/12/2016")
date2.getDate()
```

[90]:
```
'15-12-2016'
```

Class Methods with `@classmethod`

In Python, a **class method** is created with the `@classmethod` decorator and requires the class itself as the first argument, which is written as `cls`. A class method returns an instance of the class with supplied arguments or adds other additional functionality.

[91]:
```
class Cellphone:
    def __init__(self, brand, number):
        self.brand = brand
        self.number = number

    def get_number(self):
        return self.number
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def get_emergency_number():
        return "911"

    @classmethod
    def iphone(cls, number):
        print("An iPhone is created.")
        return cls("Apple", number)
```

[92]:
```
# create an iPhone instance using the class method
iphone = Cellphone.iphone("1112223333")
```

```
An iPhone is created.
```

[93]:
```
# call the instance method
iphone.get_number()
```

[93]:
```
'1112223333'
```

[94]:
```
# call the static method
iphone.get_emergency_number()
```

[94]:
```
'911'
```

[95]:
```
samsung1 = Cellphone('Samsung', '123456789')
```

[96]:
```
samsung1.get_number()
```

[96]:
```
'123456789'
```

[97]:
```
# the 'iphone' method cannot modify the instance 'samsung1'
samsung1.iphone('222222222')
```

```
An iPhone is created.
```

[97]:
```
<__main__.Cellphone at 0x2c1326f49d0>
```

[98]:
```
# the brand atribute of the instance was not modified by the 'iphone' method
# class method cannot modify specific instances
samsung1.brand
```

[98]:
```
'Samsung'
```

[99]:
```
# the number atribute of the instance was not modified by the 'iphone' method
samsung1.number
```

[99]:
```
'123456789'
```

In this example, `iphone()` is a class method since it is decorated with the `@classmethod` decorator and has `cls` as the first argument. It returns an instance of the `Cellphone` class with the brand preset to `'Apple'`.

Class methods are often used as **alternative constructors** beside the `__init__()` constructor method, or as **factory methods** in order to create instances based on different use cases.

This is shown in the following example. Here, the `__init__()` constructor method takes two parameters `name` and `age`. The class method `fromBirthYear()` takes class, name, and birthYear, and calculates the current age by subtracting it

from the current year. That is, it allows to create instances based on the year of birth, instead of based on the age. The reason for it is because we don't want the list of arguments in the `__init__()` method to be lengthy and confusing. Instead, we can use class methods to return a new instance based on different arguments.

Note again that the `fromBirthYear()` method takes `Person` class as the first parameter `cls`, and not an instance of the class Person via `self`. Also, this method returns `cls(name, date.today().year - birthYear)`, which is equivalent to `Person(name, date.today().year - birthYear)`.

[100]:
```
from datetime import date

# random Person
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def fromBirthYear(cls, name, birthYear):
        return cls(name, date.today().year - birthYear)

    def display(self):
        print(self.name + "'s age is: " + str(self.age))
```

[101]:
```
person1 = Person('Adam', 19)
person1.display()
```

```
Adam's age is: 19
```

[102]:
```
person2 = Person.fromBirthYear('John',  1985)
person2.display()
```

```
John's age is: 38
```

[103]:
```
# class method cannot modify specific instances
person1.fromBirthYear('John', 1985)
```

[103]:
```
<__main__.Person at 0x2c1326e97c0>
```

[104]:
```
person1.name
```

[104]:
```
'Adam'
```

The main difference between a static method and a class method is:

- Static methods can neither modify the class nor class instances, and they just handle the attributes. They are used to create helper or utility functions. Static methods have a logical connection with the class but do not have access to class or instance states.
- Class methods can modify the class since its parameter is always the class itself, but they cannot modify class instances. They can be used as factory methods to create new instances based on alternative information about a class.

Abstract Classes

An **abstract class** is one that never expects to be instantiated. In the next example, we will never instantiate an Animal object, but only Dog and Cat objects will be derived from the class Animal.

Abstract classes allow to create a set of methods that must be created within any subclasses built from the abstract class. An **abstract method** is a method that has a declaration but does not have an implementation. An example is the `speak` method in the Animal class. Abstract classes are helpful when designing large functional units and we want to provide a common interface for different implementations of a method.

> An **abstract class** is one that is not expected to be instantiated, and contains one or more abstract methods.

Python supports abstract classes through the `abc` module, which provides the infrastructure for defining abstract classes. Defining an abstract method is achieved by using the `@abstractmethod` decorator.

[105]:
```
from abc import ABC, abstractmethod

# Abstract class
class Animal(ABC):

    def __init__(self, name):
        self.name = name

    @abstractmethod
    def speak(self):              # Abstract method, it is not implemented
        pass

# Subclass of Animal
class Dog(Animal):

    def speak(self):
        return self.name+' says Woof!'

# Subclass of Animal
class Cat(Animal):

    def speak(self):
        return self.name+' says Meow!'

# Create instances
fido = Dog('Fido')
isis = Cat('Isis')

# The method 'speak' has different implementations for the subclasses Dog and Cat
print(fido.speak())
print(isis.speak())
```

```
Fido says Woof!
Isis says Meow!
```

Note that the abstract class Animal cannot be instantiated, because it has only an abstract version of the `speak` method.

[106]:
```
a = Animal('fido')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
~\AppData\Local\Temp\ipykernel_32936\3885960488.py in <module>
----> 1 a = Animal('fido')

TypeError: Can't instantiate abstract class Animal with abstract method speak
```

By defining an abstract class, one can define common methods for a set of subclasses. This capability is especially useful in situations where a third-party is going to provide implementations, or it can also help when working in a large team or with a large code-base where maintaining all classes is difficult or not possible.

One more example is shown, where the abstract class `Employee` has an abstract method `get_salary`. The subclasses `FulltimeEmployee` and `HourlyEmployee` are derived, and they define different `get_salary` methods for each class. The class `Payroll` has methods to add an employee and print the name and salary information.

[107]:
```
from abc import ABC, abstractmethod

# Abstract class Employee
class Employee(ABC):
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @abstractmethod
    def get_salary(self):
        pass

# Subclass
class FulltimeEmployee(Employee):
    def __init__(self, first_name, last_name, salary):
        super().__init__(first_name, last_name)
        self.salary = salary

    def get_salary(self):
        return self.salary

# Subclass
class HourlyEmployee(Employee):
    def __init__(self, first_name, last_name, worked_hours, rate):
        super().__init__(first_name, last_name)
        self.worked_hours = worked_hours
        self.rate = rate

    def get_salary(self):
        return self.worked_hours * self.rate

# A separate class Payroll
class Payroll:
    def __init__(self):
        self.employee_list = []

    def add(self, employee):
        self.employee_list.append(employee)

    def display(self):
```

(continues on next page)

(continued from previous page)

```
        for e in self.employee_list:
            print(f'{e.first_name} {e.last_name} \t ${e.get_salary()}')
```

[108]:
```
payroll = Payroll()

payroll.add(FulltimeEmployee('John', 'Doe', 6000))
payroll.add(FulltimeEmployee('Jane', 'Doe', 6500))
payroll.add(HourlyEmployee('Jenifer', 'Smith', 200, 50))
payroll.add(HourlyEmployee('David', 'Wilson', 150, 100))
payroll.add(HourlyEmployee('Kevin', 'Miller', 100, 150))
```

[109]:
```
payroll.display()
```

```
John Doe          $6000
Jane Doe          $6500
Jenifer Smith     $10000
David Wilson      $15000
Kevin Miller      $15000
```

Additional Special Methods

Indexing and Slicing with `__getitem__()` and `__setitem__()`

Indexing in Python is implemented with the built-in method `__getitem__()`.

[110]:
```
list1 = [1, 2, 3]
list1[0]
```

[110]:
```
1
```

[111]:
```
list1.__getitem__(0)
```

[111]:
```
1
```

We can implement the special method `__getitem__()` in our classes to provide built-in indexing behaviors of Python sequences to our class instances.

In the following code, the `index` argument is used to specify the elements in `self.pay`.

[112]:
```
class Employee:

    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    def __getitem__(self, index):
        return self.pay[index]
```

[113]:
```
bob = Employee(name='Bob Smith', pay=[50000, 55000, 53000, 60000])
print(bob.name, bob.pay)
```

```
Bob Smith [50000, 55000, 53000, 60000]
```

[114]:
```
bob[1]
```

[114]:
```
55000
```

[115]:
```
bob[-1]
```

[115]:
```
60000
```

Interestingly, in addition to indexing, `__getitem__()` is also used for **slicing expressions**, as shown below. The Python slice object is used for this purpose to define the starting and stopping index (and optional index step) for extracting the elements from a sequence.

[116]:
```
list1 = [1, 2, 3, 4, 5]
```

[117]:
```
list1[slice(2, 4)]
```

[117]:
```
[3, 4]
```

[118]:
```
list1[slice(1, -1)]
```

[118]:
```
[2, 3, 4]
```

[119]:
```
list1[slice(3, None)]
```

[119]:
```
[4, 5]
```

This means that we can use the `__getitem__()` method within our defined class to perform slicing, if we wanted to, and not only indexing.

[120]:
```
print(bob.name, bob.pay)
```

```
Bob Smith [50000, 55000, 53000, 60000]
```

[121]:
```
bob[0:2]
```

[121]:
```
[50000, 55000]
```

[122]:
```
bob[1:]
```

[122]:
```
[55000, 53000, 60000]
```

On the other hand, if we want to change the value of `bob.pay` outside of the class, we won't be able to do that.

[123]:
```
bob[0] = 25000
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_32936\64624401.py in <module>
----> 1 bob[0] = 25000

TypeError: 'Employee' object does not support item assignment
```

The special method `__setitem__()` allows to assign values to sequence objects. In the example below, new value can be assigned to the elements of the instance `bob` with a user-entered index.

[124]:
```
class Employee:

    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    def __setitem__(self, index, value):
        self.pay[index] = value
```

[125]:
```
bob = Employee(name='Bob Smith', pay=[50000, 55000, 53000, 60000])
print(bob.name, bob.pay)
```

```
Bob Smith [50000, 55000, 53000, 60000]
```

[126]:
```
bob[0] = 45000
```

[127]:
```
print(bob.name, bob.pay)
```

```
Bob Smith [45000, 55000, 53000, 60000]
```

Printing using `__str__()` and `__repr__()`

We know that `str()` is used to convert an object to a `str` object. Internally, it is implemented by the `__str__()` method. Moreover, Python uses `__str__()` when we call `print()` to display an object.

Let's consider again the instance of the `Employee` class. If we call the instance `bob` which we created before or if we try to print it, we can see a general Python output that tells us that it is an object created by the `Employee` class and Python also provides its memory address.

[128]:
```
bob
```

[128]:
```
<__main__.Employee at 0x2c1326f8040>
```

[129]:
```
print(bob)
```

```
<__main__.Employee object at 0x000002C1326F8040>
```

We can implement the `__str__()` method, so that, when the class instance `self` is printed, we can customize the displayed output. The code below instructs the output to list the employee name and pay. Recall the string formatting methods, which we covered earlier: `%s` with `% (name)` formatting, `{}` with `.format(name)` formatting, and f strings `f` with `{name}` formatting.

[130]:
```
class Employee:
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    def __str__(self):
        return 'Employee name %s and pay %s' % (self.name, self.pay)
        #  return 'Employee name {0} and pay {1}'.format(self.name, self.pay)
        # return f'Employee name {self.name} and pay {self.pay}'
```

[131]:
```
bob = Employee(name='Bob Smith', pay=50000)
print(bob.name, bob.pay)
```
```
Bob Smith 50000
```

[132]:
```
print(bob)
```
```
Employee name Bob Smith and pay 50000
```

[133]:
```
sue = Employee(name='Sue Jones', pay=60000)
print(sue)
```
```
Employee name Sue Jones and pay 60000
```

If we enter only the instance name without `print`, we will still obtain the general Python output.

[134]:
```
sue
```
[134]:
```
<__main__.Employee at 0x2c1326f8a00>
```

Besides the `__str__()` method, another similar method is `__repr__()` which stands for *representation* and it is also used for overloading the `print` method in Python. It provides another way to customize the printed outputs, and returns what is known as *formal string representation*. Similarly, the `__str__()` method is known as *informal string representation*.

[135]:
```
class Employee:
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    def __repr__(self):
        return '<The name and pay for the employee are {} and {} dollars>'.format(self.
↪name, self.pay)
```

[136]:
```
sue = Employee(name='Sue Jones', pay=60000)
print(sue)
```
```
<The name and pay for the employee are Sue Jones and 60000 dollars>
```

Note also in the cell below that the displayed output for `sue` is the same as for `print(sue)`. I.e., Python uses `__repr__()` to display the object in the interactive prompt.

[137]:
```
sue
```
[137]:
```
<The name and pay for the employee are Sue Jones and 60000 dollars>
```

The method `__repr__()` is more general than `_str__()` and it applies to nested appearances and a few other additional cases. The `__str__()` and `__repr__()` methods are very useful, because when other people are using our codes, they can get a good idea of what an object is by just printing it.

7.5.7 References

1. Mark Lutz, "Learning Python," 5th edition, O-Reilly, 2013. ISBN: 978-1-449-35573-9.
2. Pierian Data Inc., "Complete Python 3 Bootcamp," codes available at: link.
3. Leodanis Pozo Ramos, Python Classes: The Power of Object-Oriented Programming, available at: link
4. Python - Made with ML, Goku Mohandas, codes available at: link.
5. Jeff Knupp, Improve Your Python: Python Classes and Object Oriented Programming, available at link.
6. Python Tutorial, Python Abstract Classes, available at link.
7. Kyle Stratis, Supercharge Your Classes with Python super(), available at link.

BACK TO TOP

7.6 Lecture 6 - Exception Coding, Modules and Packages

7.6.1 6.1 Exception Coding

Most Python codes contain errors when initially developed. For example, in the second cell below, in the `print` function we used the name `Var2` instead of the name of the defined variable `var2`. When we tried to run the cell, we got a `NameError`, with the further description that `name 'Var2' is not defined`. This message is specific enough for us to realize that we used a name in the print function that is different than the name we defined.

[1]:
```
var1 = 5
print(var1)
```

```
5
```

[2]:
```
var2 = 6
print(Var2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[2], line 2
      1 var2 = 6
----> 2 print(Var2)

NameError: name 'Var2' is not defined
```

Errors detected during code execution are called **exceptions**. In this example, `NameError` is an exception. When an exception occurs, the Python interpreter terminates the program, and an error is displayed.

Errors in Python are displayed in a specific form that provides: the traceback, the type of the exception, and the error message. **Traceback** is the sequence of function calls that led to the error. In the above example, the arrow indicates that the exception occurred in line 2 of the cell. This example is extremely simple, and in actual programs the traceback will list all modules and functions which led to the exception. Most often, you can just pay attention to the last level in the traceback, which is the actual place where the error occurred.

The `try/except/else` Statement

To handle exceptions in our programs, we can use `try` and `except`. This is also known as **catching the exception**. In the following cell, the code that can cause an exception to occur is indented under the `try` header, and the `NameError` is listed after `except`. If the exception occurs, the block indented under `except` is executed. Notice that this time the cell ran despite the error in our code, and we only printed a statement.

[3]:
```
try:
    var2 = 6
    print(Var2)
except NameError:
    print('Oops, something went wrong!')
```

```
Oops, something went wrong!
```

If the `try` part succeeds (i.e., there are no errors in the block of indented statements under `try`), then the `except` part is not executed.

[4]:
```
try:
    var2 = 6
    print(var2)
except NameError:
    print('Oops, something went wrong!')
```

```
6
```

Similarly, if we try adding an integer number and a string, this will result in a `TypeError`.

[5]:
```
123 + 'abc'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[5], line 1
----> 1 123 + 'abc'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

We can again use `try` and `except` to catch this exception, only this time we use `TypeError` instead of `NameError` after the `except` keyword.

[6]:
```
try:
    123 + 'abc'
except TypeError:
     print('Oops, something went wrong!')
```

```
Oops, something went wrong!
```

What if we used the `NameError` in the `except` statement instead of `TypeError`? The result is shown below. The exception was not caught this time, because when Python executes the `try` block, it tries to match the exception type

with those listed in the `except` clause. This means that we always need to use the correct exception type in order to be caught.

[7]:
```
try:
    123 + 'abc'
except NameError:
     print('Oops, something went wrong!')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[7], line 2
      1 try:
----> 2     123 + 'abc'
      3 except NameError:
      4      print('Oops, something went wrong!')

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

But then, what if we are not sure about the type of exception that we expect to occur in our code? One solution is to except for both `NameError` and `TypeError`. This way, we can catch either a `NameError` or a `TypeError` exception.

[8]:
```
try:
    123 + 'abc'
except NameError:
    print('Oops, wrong name error!')
except TypeError:
    print('Oops, wrong type error!')
```

```
Oops, wrong type error!
```

[9]:
```
try:
    var2 = 6
    print(Var2)
except NameError:
    print('Oops, wrong name error!')
except TypeError:
    print('Oops, wrong type error!')
```

```
Oops, wrong name error!
```

Python allows to insert multiple `except` statements under a single `try` statement for catching different exception types.

It is also possible to catch any of multiple exceptions by providing a tuple of exception types after the `except` keyword, as shown in the following example.

[10]:
```
try:
    123 + 'abc'
except (NameError, TypeError):
    print('Oops, wrong name or wrong type error!')
```

```
Oops, wrong name or wrong type error!
```

When there are multiple `except` statements, the `try` block is executed line by line until the first matching exception is caught. In this example, that is the `NameError` exception, and the print line under `except NameError` is executed. The error in the line `123 + 'abc'` is not caught because the execution of the `try` block is interrupted after the first exception is detected.

[11]:
```
try:
    var2 = 6
    print(Var2)
    123 + 'abc'
except TypeError:
    print('Oops, wrong type error!')
except SyntaxError:
    print('Oops, wrong syntax error!')
except NameError:
    print('Oops, wrong name error!')
except (IndexError, IndentationError):
    print('Oops, wrong index or indentation error!')
```

```
Oops, wrong name error!
```

Another alternative is to write only `except` without specifying any exception type. An empty `except` clause will catch all exception types, and with that, we don't need to list the expected error types in the code.

[12]:
```
try:
    var2 = 6
    print(Var2)
except:
    print('Oops, something went wrong!')
```

```
Oops, something went wrong!
```

Despite this convenience, it is not generally recommended to use the empty `except` statement very often. One reason is that in the previous example we will only know that something was wrong with our code, but we won't know what caused the error. This makes fixing the program difficult. In addition, the empty `except` statement can also catch some system errors that are not related to our code (such as system exit, Ctrl+C interrupt). And even worse, it may also catch genuine programming mistakes in our code for which we probably want to see an error message.

Therefore, it is better to be specific about what types of exceptions we want to catch and where, instead of catching everything we can in the whole program.

Similarly, writing `Exception` after the `except` statement will catch all exceptions, and acts the same as an empty `except` clause. Differently from an empty `except` clause, the `Exception` statement does not catch system-related exceptions, and it is therefore somewhat preferred, but it should still be used with caution.

[13]:
```
try:
    var2 = 6
    print(Var2)
except Exception:
    print('Oops, something went wrong!')
```

```
Oops, something went wrong!
```

It is also possible to catch an exception and store it in a variable. In the following cell, we are catching an exception and storing it in the variable `my_error`.

[14]:
```
try:
    123 + 'abc'
except TypeError as var3:
    my_error = var3
```

[15]:
```
my_error
```

[15]:
```
TypeError("unsupported operand type(s) for +: 'int' and 'str'")
```

The syntax of the `try/except` statement can also include an optional `else` statement. The block of statements indented under `else` is executed if there is no exception caught in the `try` block.

In this example, there is an exception in the `print(Var2)` line, and because of that, the statement under `except` is executed.

[16]:
```
try:
    var2 = 6
    print(Var2)
    print('This message is not printed')
except NameError:
    print('Oops, something went wrong!')
else:
    print('The code is executed successfully, no exception occurred!')
```

```
Oops, something went wrong!
```

On the contrary, the following code does not raise an exception, and therefore, the statements under `try` are executed, and also, the statement under `else` is executed.

[17]:
```
try:
    var2 = 6
    print(var2)
    print('This message is printed')
except NameError:
    print('Oops, something went wrong!')
else:
    print('The code is executed successfully, no exception occurred!')
```

```
6
This message is printed
The code is executed successfully, no exception occurred!
```

The general syntax of the `try/except/else` statement is as shown below. It is a compound, multipart statement, that starts with a `try` header. It is followed by one or more `except` blocks, which identify exceptions to be caught and blocks to process them. The `else` statement is optional, and it is listed after the `except` blocks; the `else` block runs if no exceptions are encountered. The words `try`, `except`, and `else` should be indented to the same level (vertically aligned).

```
try:
   Place your operations here.
   ...
   ...
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
except (ExceptionIII, ExceptionIV):
    If there is ExceptionIII or ExceptionIV, then execute this block.
except ExceptionV as Var1:
    If there is ExceptionV, store it in the variable Var1, and then execute this block.
```

(continues on next page)

(continued from previous page)

```
except:
    If there are any other exceptions, then execute this block.
   ...
   ...
else:
   If there is no exception, then execute this block.
```

The `finally` Statement

The `finally` statement is another statement that can be combined with `try`. The general syntax is shown below. The goal is to always execute the block of code indented under `finally` regardless of whether there was an exception in the `try` block or not.

```
try:
   Place your operations here
   ...
   ...
   Due to exceptions, these lines of code may be skipped.
finally:
   This code block is always executed, regardless of whether exceptions occurred.
```

The `try/finally` form is useful when we want to be completely sure that an action will happen after some code runs, without considering the exception behavior of the program. In practice, this allows to specify cleanup actions that must always occur, such as file closes or server disconnects.

The next example opens a file named `testfile` for writing, then writes some text, and closes the file. The code under `finally` is executed.

[18]:
```
try:
    f = open('testfile', 'w')
    f.write('First sentence, second sentence, end')
    f.close()
finally:
    print('The finally code block is always executed')
```

```
The finally code block is always executed
```

Then, this cell reads the file.

[19]:
```
try:
    f = open('testfile', 'r')
    print(f.read())
    f.close()
finally:
    print('The finally code block is always executed')
```

```
First sentence, second sentence, end
The finally code block is always executed
```

For practice, let's make an intentional mistake and try to open a file for reading that does not exist. As expected, we got a `FileNotFoundError`, however the code under `finally` was still executed.

[20]:
```
try:
    f = open('wrongfile', 'r')
    print(f.read())
    f.close()
finally:
    print('The finally code block is always executed')
```

```
The finally code block is always executed
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[20], line 2
      1 try:
----> 2     f = open('wrongfile', 'r')
      3     print(f.read())
      4     f.close()

File ~\anaconda3\Lib\site-packages\IPython\core\interactiveshell.py:284, in _modified_
↪open(file, *args, **kwargs)
    277 if file in {0, 1, 2}:
    278     raise ValueError(
    279         f"IPython won't let you open fd={file} by default "
    280         "as it is likely to crash IPython. If you know what you are doing, "
    281         "you can use builtins' open."
    282     )
--> 284 return io_open(file, *args, **kwargs)

FileNotFoundError: [Errno 2] No such file or directory: 'wrongfile'
```

The `finally` clause can also be combined with `except` and `else`. The logic remains the same, that is, the block under `finally` will always be executed. In this example the exception is caught, and the print statement under `except` and `finally` are displayed.

[21]:
```
try:
    f = open('wrongfile', 'r')
    print(f.read())
    f.close()
except FileNotFoundError:
    print('Oops, there is no such file')
finally:
    print('The finally code block is always executed')
```

```
Oops, there is no such file
The finally code block is always executed
```

Error Types

Besides the above types `NameError` and `TypeError`, let's briefly look at several other common error types in Python.

`SyntaxError` occurs when there is a problem with the structure of the code in the program (e.g., EOL below stands for End-Of-Line error, meaning that we forgot the single quote at the end of the string in this example). `IndexError` points to wrong indexing of sequences. `IndentationError`, `FileEror`, `ZeroDivisionError` are self-explanatory.

[22]:
```
print('Hello world)
```

```
  Cell In[22], line 1
    print('Hello world)
          ^
SyntaxError: unterminated string literal (detected at line 1)
```

[23]:
```
list1 = [1, 2, 3]
list1[10]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[23], line 2
      1 list1 = [1, 2, 3]
----> 2 list1[10]

IndexError: list index out of range
```

[24]:
```
def func1():
    msg = 'Hello world'
    print(msg)
     return msg
```

```
  Cell In[24], line 4
    return msg
    ^
IndentationError: unexpected indent
```

[25]:
```
myfile = open('newfile.txt', 'r')
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[25], line 1
----> 1 myfile = open('newfile.txt', 'r')

File ~\anaconda3\Lib\site-packages\IPython\core\interactiveshell.py:284, in _modified_
↪open(file, *args, **kwargs)
    277 if file in {0, 1, 2}:
    278     raise ValueError(
    279         f"IPython won't let you open fd={file} by default "
    280         "as it is likely to crash IPython. If you know what you are doing, "
    281         "you can use builtins' open."
    282     )
--> 284 return io_open(file, *args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
FileNotFoundError: [Errno 2] No such file or directory: 'newfile.txt'
```

[26]:
```
10/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[26], line 1
----> 1 10/0

ZeroDivisionError: division by zero
```

[27]:
```
# The zero division error is not detected because the indentation error was first␣
↪detected and the line didn't run
x = 0
 print(10/x)
```

```
  Cell In[27], line 3
    print(10/x)
    ^
IndentationError: unexpected indent
```

All exceptions in Python are shown below. Detailed explanations about each exception can be found here.

Note that the exceptions have a hierarchy, where for instance, catching an `ArithmeticError` exception will catch everything that is under it in the tree, i.e., `FloatingPointError`, `OverflowError`, and `ZeroDivisionError`. There are also a few exceptions that are not in this tree, like `SystemExit` and `KeyboardInterrupt`, but most of the time we shouldn't catch these exceptions.

```
Exception
├── ArithmeticError
│   ├── FloatingPointError
│   ├── OverflowError
│   └── ZeroDivisionError
├── AssertionError
├── AttributeError
├── BufferError
├── EOFError
├── ImportError
├── LookupError
│   ├── IndexError
│   └── KeyError
├── MemoryError
├── NameError
│   └── UnboundLocalError
├── OSError
│   ├── BlockingIOError
│   ├── ChildProcessError
│   ├── ConnectionError
│   │   ├── BrokenPipeError
│   │   ├── ConnectionAbortedError
│   │   ├── ConnectionRefusedError
```

(continues on next page)

(continued from previous page)

```
    │   └── ConnectionResetError
    ├── FileExistsError
    ├── FileNotFoundError
    ├── InterruptedError
    ├── IsADirectoryError
    ├── NotADirectoryError
    ├── PermissionError
    ├── ProcessLookupError
    └── TimeoutError
├── ReferenceError
├── RuntimeError
│   └── NotImplementedError
├── StopIteration
├── SyntaxError
│   └── IndentationError
│       └── TabError
├── SystemError
├── TypeError
├── ValueError
│   └── UnicodeError
│       ├── UnicodeDecodeError
│       ├── UnicodeEncodeError
│       └── UnicodeTranslateError
└── Warning
    ├── BytesWarning
    ├── DeprecationWarning
    ├── FutureWarning
    ├── ImportWarning
    ├── PendingDeprecationWarning
    ├── ResourceWarning
    ├── RuntimeWarning
    ├── SyntaxWarning
    ├── UnicodeWarning
    └── UserWarning
```

The `raise` Statement

In Python, we can also trigger exceptions and create error messages manually. This is known as **raising an exception**, and it is coded with the `raise` keyword followed by the exception and an optional error message.

The general syntax is as follows:

```
if test_condition:
    raise Exception(Message)
```

In the following example, we raise an exception and stop the program if `x` is less than 0. In the parentheses, we specified the text that is to be displayed in the error message.

[28]:
```
x = -1

if x < 0:
    raise Exception('Sorry, no numbers below zero')
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
Cell In[28], line 4
      1 x = -1
      3 if x < 0:
----> 4     raise Exception('Sorry, no numbers below zero')

Exception: Sorry, no numbers below zero
```

We can also define the type of exception to raise after the `raise` keyword, such as `TypeError` in the next example.

[29]:
```
y = 'hello'

type(y)
```

[29]:
```
str
```

[30]:
```
if type(y) is not int:
    raise TypeError('Only integers are allowed')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[30], line 2
      1 if type(y) is not int:
----> 2     raise TypeError('Only integers are allowed')

TypeError: Only integers are allowed
```

When an exception is not raised, the indented block under `raise` is not executed.

[31]:
```
y = 3

if type(y) is not int:
    raise TypeError('Only integers are allowed')
```

We can also create custom exceptions ahead of time, and use them afterward in our code.

[32]:
```
my_exception = TypeError('Sorry, the input should be an integer number')

z = 'one'

if type(z) is not int:
    raise my_exception
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[32], line 6
      3 z = 'one'
      5 if type(z) is not int:
----> 6     raise my_exception

TypeError: Sorry, the input should be an integer number
```

In the above cell, `my_exception` is in fact an instance of the class `TypeError`. The error message that we typed above is an attribute of the created instance of `TypeError` class.

Additionally, the `raise` statement can be used alone, without an exception name. In that case, it simply reraises the current exception. This form is typically used if we need to catch and handle an exception, but don't want the exception to be hidden and terminated in the code.

Consider the following example where `except` catches a `ZeroDivisionError`.

[33]:
```
a = 10
b = 0
try:
    print(a/b)
except ZeroDivisionError:
    print('Oops, something went wrong')
```

```
Oops, something went wrong
```

Including the `raise` statement alone at the end of the code causes the exception to be reraised.

[34]:
```
a = 10
b = 0
try:
    print(a/b)
except ZeroDivisionError:
    print('Oops, something went wrong')
    raise
```

```
Oops, something went wrong
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[34], line 4
      2 b = 0
      3 try:
----> 4     print(a/b)
      5 except ZeroDivisionError:
      6     print('Oops, something went wrong')

ZeroDivisionError: division by zero
```

Here is another example, where the function `quad` is used for calculating the roots of a quadratic function with coefficients `a`, `b`, and `c`. The user-defined `QuadError` raises an exception if the function is not quadratic, or if it does not have real roots. The `raise` statement allows us to introduce application-specific errors in our codes.

[35]:
```
import math
class QuadError(Exception): pass

def quad(a, b, c):
    if a == 0:
        raise QuadError('Not quadratic')

    if b*b-4*a*c < 0:
        raise QuadError('No real roots')

    x1 = (-b+math.sqrt(b*b-4*a*c))/(2*a)
    x2 = (-b-math.sqrt(b*b-4*a*c))/(2*a)

    return (x1, x2)
```

[36]:
```
x1, x2 = quad(3, 4, 4)
print("Roots are", x1, x2)
```

```
---------------------------------------------------------------------------
QuadError                                 Traceback (most recent call last)
Cell In[36], line 1
----> 1 x1, x2 = quad(3, 4, 4)
      2 print("Roots are", x1, x2)

Cell In[35], line 9, in quad(a, b, c)
      6     raise QuadError('Not quadratic')
      8 if b*b-4*a*c < 0:
----> 9     raise QuadError('No real roots')
     11 x1 = (-b+math.sqrt(b*b-4*a*c))/(2*a)
     12 x2 = (-b-math.sqrt(b*b-4*a*c))/(2*a)

QuadError: No real roots
```

[37]:
```
x1, x2 = quad(1, -5, 6)
print("Roots are", x1, x2)
```

```
Roots are 3.0 2.0
```

[38]:
```
x1, x2 = quad(0, -5, 6)
print("Roots are", x1, x2)
```

```
---------------------------------------------------------------------------
QuadError                                 Traceback (most recent call last)
Cell In[38], line 1
----> 1 x1, x2 = quad(0, -5, 6)
      2 print("Roots are", x1, x2)

Cell In[35], line 6, in quad(a, b, c)
      4 def quad(a, b, c):
      5     if a == 0:
----> 6         raise QuadError('Not quadratic')
      8     if b*b-4*a*c < 0:
      9         raise QuadError('No real roots')

QuadError: Not quadratic
```

The `assert` Statement

The `assert` statement is similar to the `raise` statement, and it can be thought of as a *conditional* `raise` statement.

The general syntax is:

```
assert test_condition, message(optional)
```

If the `test condition` evaluates to False, Python raises an `AssertionError` exception. If the `message` item is provided, it is used as the error message in the displayed exception.

Conversely, if the `test condition` evaluates to True, the program will continue to the next line and will do nothing.

Like all exceptions, the `AssertionError` exception will terminate the program if it's not caught with a `try` statement.

In the following example, `assert` is used to ensure that the values for the `Temperature` are non-negative.

[39]:
```
def KelvinToFahrenheit(Temperature):
    assert Temperature >= 0, 'Colder than absolute zero!'
    return ((Temperature-273)*1.8)+32
```

[40]:
```
KelvinToFahrenheit(273)
```

[40]:
```
32.0
```

[41]:
```
KelvinToFahrenheit(-10)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[41], line 1
----> 1 KelvinToFahrenheit(-10)

Cell In[39], line 2, in KelvinToFahrenheit(Temperature)
      1 def KelvinToFahrenheit(Temperature):
----> 2     assert Temperature >= 0, 'Colder than absolute zero!'
      3     return ((Temperature-273)*1.8)+32

AssertionError: Colder than absolute zero!
```

The equivalent `assert` code of the next cell using the `raise` statement is shown in the cell below.

[42]:
```
x = -1

if x < 0:
    raise Exception('Sorry, no numbers below zero')
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
Cell In[42], line 4
      1 x = -1
      3 if x < 0:
----> 4     raise Exception('Sorry, no numbers below zero')

Exception: Sorry, no numbers below zero
```

[43]:
```
x = -3

assert x >= 0, 'Sorry, no numbers below zero'
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[43], line 3
      1 x = -3
----> 3 assert x >= 0, 'Sorry, no numbers below zero'

AssertionError: Sorry, no numbers below zero
```

Here is one more simple example, where `assert` is used to ensure that no empty lists are passed to `marks`.

[44]:
```
def average(marks):
    assert len(marks) != 0, 'List is empty'
    return sum(marks)/len(marks)
```

[45]:
```
marks1 = [55, 88, 78, 90, 79]
print('Average of marks is:', average(marks1))
```

```
Average of marks is: 78.0
```

[46]:
```
marks2 = []
print('Average of marks is:', average(marks2))
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[46], line 2
      1 marks2 = []
----> 2 print('Average of marks is:', average(marks2))

Cell In[44], line 2, in average(marks)
      1 def average(marks):
----> 2     assert len(marks) != 0, 'List is empty'
      3     return sum(marks)/len(marks)

AssertionError: List is empty
```

`Assert` is typically used to verify program conditions during development (such as user-defined constraints), rather than for catching genuine programming errors. Because Python catches programming errors itself, there is usually no need to use `assert` to catch things like zero divides, out-of-bounds indexes, type mismatches, etc. In general, assertions are useful for checking types, classes, or values of inputted variables, checking data structures such as duplicates in a list or contradictory variables, and checking that outputs of functions are reasonable and as expected.

7.6.2 6.2 Module Coding Basics

Every Python file with code is referred to as a **module**. To create modules, we don't need to write special syntax to tell Python that we are making a module. We can simply use any text editor to type Python code into a text file, and save it with a `.py` extension; any such file is automatically considered a Python module.

For example, I have created a simple file called `my_module.py` that is saved in the same directory as this Jupyter notebook. The module does not do anything useful, it just defines a few names and prints a few statements. The code inside `my_module.py` is shown below.

my_module.py

```
1  print('I am inside my_module')
2
3  X = 3
4  print('The value of the variable X is:', X)
5
6  Y = 5
7
8  def sub_report():
9      Z = 8
10     print('The value of the variable Z is:', Z)
11     print('I am a function named sub_report')
12
13 def main_report():
14     U = 10
15     print('The value of the variable U is:', U)
16     print('I am a function named main_report')
17
```

Similar to the rules for naming other variables in Python, module names should follow the same rules and can contain only letters, digits, and underscores. The module names cannot use Python-reserved keywords (e.g., such as a module file named `if.py`.)

The `import` Statement

Python programs can use the modules file we have created by running an `import` or `from` statement. These statements find, compile, and run a module file's code. The main difference is that `import` fetches the module as a whole, while `from` fetches specific names out of the module.

Let's import `my_module`. Python executes the statements in the module file one after another, from the top of the file to the bottom. For this module, the two print statements at the top level of the file are executed. The print statements inside the two functions (`main_report` and `sub_report`) are not executed; they will be executed only when the functions `sub_report` and `main_report` are called.

[47]:
```
import my_module
```

```
I am inside my_module
The value of the variable X is: 3
```

Note that we don't use the `.py` extension for the files with the `import` statement (i.e., `import my_module.py` will raise an exception).

When the module is imported, a new **module object** is created. The module object is shown below, where Python mapped the module name to an external filename by adding a directory path from the module search path to the file, and a `.py` extension at the end.

[48]:
```
# The name my_module references to the loaded module object
my_module
```

[48]:
```
<module 'my_module' from 'C:\\Users\\vakanski\\Documents\\Codes\\2023 Codes\\Python for Data Science Course\\Lectures_2023\\Theme_1-Python_Programming\\Lecture_6-Exceptions,_Modules\\Posted\\Lecture_6-Exceptions,_Modules\\my_module.py'>
```

Overall, the name `my_module` serves two different purposes: 1. It identifies the external file `my_module.py` that needs to be loaded. 2. After the module is loaded, it becomes a reference to the module object.

During importing, all the names assigned at the top level of the module become attributes of the module object. In this example, the variables `X` and `Y` and the functions `sub-report` and `main_report` become attributes of the module, and we can call them by using the `object.attribute` syntax (a.k.a. *qualification*).

[49]:
```
my_module.X
```

[49]:
```
3
```

[50]:
```
my_module.Y
```

[50]:
```
5
```

[51]:
```
my_module.sub_report()
```

```
The value of the variable Z is: 8
I am a function named sub_report
```

The `from` Statement

The `from` statement fetches specific names from the module, and allows to use the names directly (without the need for `module_object.attribute`). This way, we can call the names in the module with less typing.

[52]:
```
from my_module import X
X
```

[52]:
```
3
```

The `from` statement in effect copies the names out of the module into another scope; in this case, in the scope of this Jupyter notebook, where the `from` statement appears.

When we run a `from` statement, internally Python first imports the entire module file as usual, then copies the specific names out of the module file, and finally, it deletes the module file. This is similar to the following code:

```
import my_module
X = module.X
del my_module
```

With `from`, we can also import several names at the same time, separated by commas.

[53]:
```
from my_module import X, Y, sub_report
```

[54]:
```
sub_report()
```

```
The value of the variable Z is: 8
I am a function named sub_report
```

Another alternative is to use a * instead of specific names, which fetches all names assigned at the top level of the referenced module. The following code fetches all four names in our module: `X`, `Y`, `sub_report`, and `main_report`. Note again that the names `Z` and `U` are not defined at the top level in the module, but are enclosed in the functions, and therefore, they can not be fetched with the `import` statement.

[55]:
```
from my_module import *
main_report()
```

```
The value of the variable U is: 10
I am a function named main_report
```

[56]:
```
U
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[56], line 1
----> 1 U

NameError: name 'U' is not defined
```

One problem with using `from module import *` is that it can silently overwrite variables that have the same name as existing variables in our scope.

In the following example, we have a variable `X = 15`, which was overwritten by the variable `X` with the same name in `my_module` which has the value 3. The way this variable was overwritten may not be obvious (e.g., in large modules with many variables we cannot remember and keep track of all variable names).

[57]:
```
X = 15
from my_module import *
print(X)
```

```
3
```

On the other hand, if we use `import`, all names will be defined only within the scope of the module, and the names will not collide with other names in our programs.

[58]:
```
X = 15
import my_module
# The print statements this time were not displayed (the reason why it so is explained␣
↪in the Appendix)
```

[59]:
```
print(X)
print(my_module.X)
```

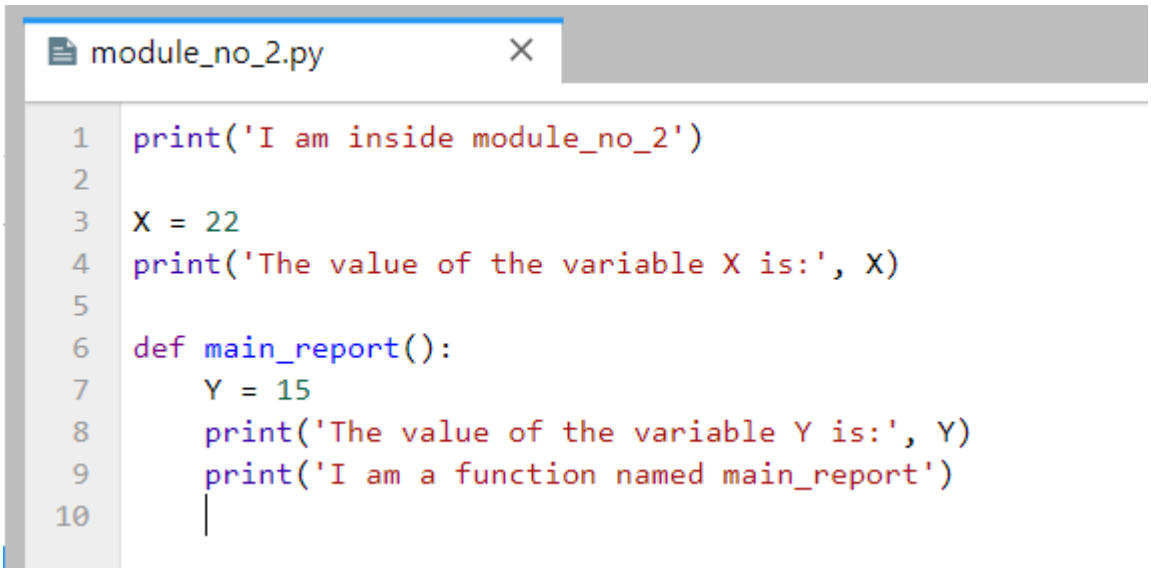
```
15
3
```

Therefore, programmers need to be careful when using the `from` statement (especially with *), and the `import` only statement should be preferred. However, `from` also provides convenience of less typing, and it is still very commonly used.

When Using `import` is Required

When the same name of a variable or function is defined in two different modules, and we need to use both of the names at the same time, then we must use the `import` statement.

For instance, let's assume that another module file named `module_no_2.py` also contains a variable `X` and a function `main_report`.



Using `import` we can load the two different variables `X`, because including the name of the enclosing module makes the two names unique.

[60]:
```
import my_module # when a module is imported the first time, it is executed
import module_no_2 # when a module is imported afterward, it is not executed
```

```
I am inside module_no_2
The value of the variable X is: 22
```

[61]:
```
print(my_module.X)
print(module_no_2.X)
```

```
3
22
```

The same holds for the function `main_report` which appears in both modules.

[62]:
```
my_module.main_report()
module_no_2.main_report()
```

```
The value of the variable U is: 10
I am a function named main_report
The value of the variable Y is: 15
I am a function named main_report
```

In this case, the `from` statement will fail because we can have only one assignment to the name `X` in the scope.

[63]:
```
# Only one variable name X can exist at one time
from my_module import X
```

(continues on next page)

(continued from previous page)

```
from module_no_2 import X
print(X)
```

```
22
```

Another way to resolve the name clashing problem is to use the `as` extension to `from/import` that allows to import a name under another name that will be used as a synonym.

[64]:
```
from my_module import X as X1
from module_no_2 import X as X2
print(X1)
print(X2)
```

```
3
22
```

Module Namespaces

Modules can be understood as places where collections of names are defined that we want to make visible to the rest of our code. These collections of names live in the module's namespace and represent the attributes of the module object.

To access the namespace of `my_module` object, we can use the built-in `dir` method. We can notice the names we assigned to the module file: `X`, `Y`, `main_report`, and `sub_report`. However, Python also adds some names in the module's namespace for us; for instance, `__file__` gives the path to the file the module was loaded from, and `__name__` gives the module name.

[65]:
```
dir(my_module)
```

[65]:
```
['X',
 'Y',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'main_report',
 'sub_report']
```

Internally, the module namespaces created by imports are stored as dictionary objects. Module namespaces can also be accessed through the built-in `__dict__` attribute associated with module objects, where the names are dictionary keys.

[66]:
```
my_module.__dict__.keys()
```

[66]:
```
dict_keys(['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__file__', '_
↪_cached__', '__builtins__', 'X', 'Y', 'sub_report', 'main_report'])
```

[67]:
```
my_module.__dict__['__file__']
```

[67]:
```
'C:\\Users\\vakanski\\Documents\\Codes\\2023 Codes\\Python for Data Science Course\\
↪Lectures_2023\\Theme_1-Python_Programming\\Lecture_6-Exceptions,_Modules\\Posted\\
↪Lecture_6-Exceptions,_Modules\\my_module.py'
```

[68]:
```
my_module.__dict__['__name__']
```

[68]:
```
'my_module'
```

7.6.3 6.3 Module Packages

When we create programs in Python, it is helpful to organize the individual module files related to an application into sub-directories. A directory of Python code is said to be a **package** or **modules package**. Importing a directory is known as a package import.

For example, consider the directory `MyMainPackage` which is located in the same directory as this Jupyter notebook.

```
MyMainPackage
    ├── __init__.py
    ├── main_script
    ├── MySubPackage
    │   ├── __init__.py
    │   ├── sub_script.py
```

To import the module file `sub_script.py` which is located inside the directory `MySubPackage`, we can use the dotted syntax shown in the following cell `MyMainPackage.MySubPackage.sub_script`. In effect, this turns the directory `MyMainPackage` into a Python namespace, which has attributes corresponding to the sub-directories and module files that the directory contains.

sub_script.py

```
1  print('I am inside sub_script, which is located in MySubPackage')
2
3  X = 23
4  print('The value of the variable X is:', X)
5
6  def sub_report():
7      Y = 5
8      print('I am a function inside sub_script')
9      print('The value of the variable Y is:', Y)
10
```

[69]:
```
import MyMainPackage.MySubPackage.sub_script
```

```
I am inside sub_script, which is located in MySubPackage
The value of the variable X is: 23
```

As we learned previously, `import` fetches a module as a whole, and the names (variables and functions) that are defined in the module `sub_script.py` become attributes of the imported object. These include the variable `X` and the function `sub_report`.

[70]:
```
MyMainPackage.MySubPackage.sub_script.X
```

[70]:
```
23
```

[71]:
```
MyMainPackage.MySubPackage.sub_script.sub_report()
```

```
I am a function inside sub_script
The value of the variable Y is: 5
```

The dotted path in the cell corresponds to the path through the directory hierarchy that leads to the module file `sub_script.py`, i.e., `MyMainPackage\MySubPackage\sub_script.py`.

On the other hand, note that syntax with backward slashes does not work with the `import` statement.

[72]:
```
import MyMainPackage\MySubPackage\sub_script
```

```
  Cell In[72], line 1
    import MyMainPackage\MySubPackage\sub_script
                        ^
SyntaxError: unexpected character after line continuation character
```

Similarly to `import` and `from` statements with modules, to fetch specific names from the `sub_script.py` module, we can use the `from` statement with packages as well.

[73]:
```
from MyMainPackage.MySubPackage.sub_script import X
```

[74]:
```
X
```

[74]:
```
23
```

[75]:
```
from MyMainPackage.MySubPackage.sub_script import sub_report
```

[76]:
```
sub_report()
```

```
I am a function inside sub_script
The value of the variable Y is: 5
```

Package `__init__.py` Files

When using package imports, there is one more constraint that we need to follow: each directory named within the path of a package import statement must contain a file named `__init__.py`. Otherwise, the package import will fail.

In the example we have been using, note that both `MyMainPackage` and `MySubPacakge` directories contain a file called `__init__.py`. The `__init__.py` names are special, as they declare that a directory is a Python package.

The `__init__.py` files are very often completely empty, and don't contain any code. But, they can also contain Python code, just like other module files. In our `MyMainPackage` example, the `__init__.py` files are empty.

The `__init__.py`files are run automatically the first time a Python program imports a directory. Because of that, `__init__.py` files can be used to store code to initialize the state required by files in a package (e.g., to create required data files, open connections to databases, and so on).

On a separate note, don't confuse `__init__.py` files in module packages with the `__init__()` class constructor method that we used before for specifying attributes of class instances. Both have initialization roles, but they are otherwise very different.

Difference Between `from` and `import` with Packages

The `import` statement can be somewhat inconvenient to use with packages, because we may have to retype the paths to the files and sub-directories frequently in our program. In our example, we must retype and rerun the full path from `MyMainPackage` each time we want to reach the names in the `sub_script.py` file. Otherwise, we will get an error.

[77]:
```
sub_script.X
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[77], line 1
----> 1 sub_script.X

NameError: name 'sub_script' is not defined
```

[78]:
```
MySubPackage.sub_script.X
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[78], line 1
----> 1 MySubPackage.sub_script.X

NameError: name 'MySubPackage' is not defined
```

[79]:
```
MyMainPackage.MySubPackage.sub_script.X
```

[79]:
```
23
```

[80]:
```
# Use X in our code
print(MyMainPackage.MySubPackage.sub_script.X + 27)
print(MyMainPackage.MySubPackage.sub_script.X % 2)
print((MyMainPackage.MySubPackage.sub_script.X -13)/2)
```

```
50
1
5.0
```

It is often more convenient to use the `from` statement with packages to avoid retyping the paths at each access.

[81]:
```
from MyMainPackage.MySubPackage.sub_script import X
X
```

[81]:
```
23
```

[82]:
```
print(X + 27)
print(X % 2)
print((X - 13)/2)
```

```
50
1
5.0
```

In addition, if we ever restructure or rename the directory tree, the `from` statement requires just one path update in the code, whereas the `import` statement may require updates in many lines in the code.

However, `import` can be advantageous if there are two modules with the same name that are located in different directories, and are used in the same program. With the `from` statement, we can reach only one of the two modules at

a time.

For example, in our `MyMainPackage`, there is a function `sub_report` in both the `main_script` and `sub_script`. If we use `from` statement, the name `sub_report` will change depending on whether it is imported from the `main_script` or the `sub_script`.

sub_script.py

```
1  print('I am inside sub_script, which is located in MySubPackage')
2
3  X = 23
4  print('The value of the variable X is:', X)
5
6  def sub_report():
7      Y = 5
8      print('I am a function inside sub_script')
9      print('The value of the variable Y is:', Y)
10
```

main_script.py

```
1  print('I am inside main_script, which is located in MyMainPackage')
2
3  X = 12
4  print('The value of the variable X is:', X)
5
6  def report_main():
7      Y = 10
8      print('I am a function inside main_script')
9      print('The value of the variable Y is:', Y)
10
11 def sub_report():
12     Z = 6
13     print('I am a function inside main_script')
14     print('The value of the variable Z is:', Z)
```

[83]:
```
from MyMainPackage.MySubPackage.sub_script import sub_report
```

[84]:
```
sub_report()
```

```
I am a function inside sub_script
The value of the variable Y is: 5
```

[85]:
```
from MyMainPackage.main_script import sub_report
```

```
I am inside main_script, which is located in MyMainPackage
The value of the variable X is: 12
```

[86]:
```
# Name collision with the sub_report name used in the cell above
sub_report()
```

```
I am a function inside main_script
The value of the variable Z is: 6
```

But, with the `import` statement, we can use either of the two functions `sub_report`, because their names will involve their full path, and this way, the names will not clash. The only inconvenience is that we need to type the full paths to the two functions.

[87]:
```
import MyMainPackage.MySubPackage.sub_script
MyMainPackage.MySubPackage.sub_script.sub_report()
```

```
I am a function inside sub_script
The value of the variable Y is: 5
```

[88]:
```
import MyMainPackage.main_script
MyMainPackage.main_script.sub_report()
```

```
I am a function inside main_script
The value of the variable Z is: 6
```

Another alternative is to use the `as` extension, which will create unique synonyms for the names of the two functions. As we mentioned before, this extension is commonly used to provide short synonyms for longer names, and to avoid name clashes when we are already using a name in a script that would otherwise be overwritten by a regular `import` statement.

[89]:
```
from MyMainPackage.MySubPackage.sub_script import sub_report as sub_sub_report
sub_sub_report()
```

```
I am a function inside sub_script
The value of the variable Y is: 5
```

[90]:
```
from MyMainPackage.main_script import sub_report as main_sub_report
main_sub_report()
```

```
I am a function inside main_script
The value of the variable Z is: 6
```

7.6.4 Appendix: Modules and Packages Extras

Modules Usage Modes: `__name__` and `__main__`

We mentioned that each module has a built-in attribute called `__name__`, which Python assigns automatically to all module objects. The attribute is assigned as follows: - If the file is being imported by using the `import` statement, `__name__` is set to the module's name. - If the file is being run as a top-level program file, `__name__` is set to the string `__main__`.

Let's check it with an example. The module file `module_no_3` is shown below, and note that in the first line we will print the assigned attribute `__name__` to confirm that the above is correct.

module_no_3.py

```
1  print('Print the built-in attribute name of the module:',__name__)
2
3  X = 1
4  print('The value of the variable X is:', X)
5
6  def CelsiusToFahrenheit(Temperature):
7      assert Temperature >= 0, 'Colder than absolute zero!'
8      return ((Temperature-32)/1.8)
9
```

As expected, `__name__` is assigned to `module_no_3` when imported.

[91]:
```
# The module is imported
import module_no_3
```

```
Print the built-in attribute name of the module: module_no_3
The value of the variable X is: 1
```

When `module_no_3` is run directly, `__name__` is set to `__main__`.

[92]:
```
# The module is run by passing it as a command to the Python interpreter
!python module_no_3.py
```

```
Print the built-in attribute name of the module: __main__
The value of the variable X is: 1
```

Thus, the `__name__` attribute can be used in the following `if` test `if __name__ == '__main__'` to determine whether it is being run or imported.

Therefore, if the module is the main script in a package and represents an entry point to a package that is run by the end-users (`!python mainscript.py`), the code after `if __name__ == '__main__'` in the main script will be executed when the main script is run. On the other hand, all other modules in the package will be imported. Any code under the `if __name__ == '__main__'` test in other modules will not be executed.

Another reason why using this is helpful is during code development for self-testing code that is written at the bottom of a file under the `__name__` test. For instance, the file `module_no_3a` is similar to the file `module_no_3`, only that it includes several lines of code at the bottom, which test whether the function `CelsiusToFahrenheit` outputs expected values. When run as a command in the cell, the `if __name__ == '__main__'` is True, and the lines that test the outputs of the `CelsiusToFahrenheit` are run. Conversely, when the module file is imported, the various variables and functions are imported, but the `if __name__ == '__main__'` is False, and the lines that test the outputs of the `CelsiusToFahrenheit` are not run.

module_no_3a.py

```
1  print('Print the built-in attribute name of the module:',__name__)
2
3  X = 1
4  print('The value of the variable X is:', X)
5
6  def CelsiusToFahrenheit(Temperature):
7      assert Temperature >= 0, 'Colder than absolute zero!'
8      return ((Temperature-32)/1.8)
9
10 if __name__ == '__main__':
11     print(20*'-')
12     print('Self-testing')
13     print('100 degrees Fahrenheit is', CelsiusToFahrenheit(100),'degrees Celsius')
14     print('32 degrees Fahrenheit is', CelsiusToFahrenheit(32),'degrees Celsius')
15     print('0 degrees Fahrenheit is', CelsiusToFahrenheit(0),'degrees Celsius')
16
```

[93]:
```
!python module_no_3a.py
```

```
Print the built-in attribute name of the module: __main__
The value of the variable X is: 1
--------------------
Self-testing
100 degrees Fahrenheit is 37.77777777777778 degrees Celsius
32 degrees Fahrenheit is 0.0 degrees Celsius
0 degrees Fahrenheit is -17.77777777777778 degrees Celsius
```

[94]:
```
import module_no_3a
```

```
Print the built-in attribute name of the module: module_no_3a
The value of the variable X is: 1
```

The above code allows to test the logic in our code without having to retype everything at the notebook cell or at the interactive command line each time we edit the file. Besides, the output of the self-test call will not appear every time this file is imported from another file.

Functions defined in files with the `__name__` test can be run as standalone functions, and they can also be reused in other programs.

Reloading Modules

As we have seen, when we `import` a module, the code is executed only once when the module is imported the first time. Subsequent imports use the already loaded module object without reloading or rerunning the file's code.

To force a module's code to be reloaded and rerun, you need to instruct Python to do so explicitly by calling the `reload` built-in function. The `reload` reruns a module file's code and overwrites its existing namespace, rather than deleting the module object and re-creating it. Also, the `reload` function returns the module object at the output of the cell.

[95]:
```
import my_module
```

[96]:
```
from imp import reload
reload(my_module)
```

```
I am inside my_module
The value of the variable X is: 3
```

```
C:\Users\vakanski\AppData\Local\Temp\ipykernel_18208\2249891335.py:1: DeprecationWarning:
↪ the imp module is deprecated in favour of importlib and slated for removal in Python␣
↪3.12; see the module's documentation for alternative uses
  from imp import reload
```

[96]:
```
<module 'my_module' from 'C:\\Users\\vakanski\\Documents\\Codes\\2023 Codes\\Python for␣
↪Data Science Course\\Lectures_2023\\Theme_1-Python_Programming\\Lecture_6-Exceptions,_
↪Modules\\Posted\\Lecture_6-Exceptions,_Modules\\my_module.py'>
```

Reloading can help to examine a file, for instance, when we make changes to the file. In this case, since we use Jupyter notebooks, to `import` a file again after we have made some changes to the file we can just restart the kernel, which will allow us to import the file, without using `reload`.

Module Packages Reloading

Just like module files, an already imported directory needs to be passed to `reload` to force re-execution of the code. As shown, `reload` accepts a dotted path name to reload nested directories and files. Also, `reload` returns the module object in the displayed output of the cell.

[97]:
```
# Repeated import statements do not produce any output
import MyMainPackage.MySubPackage.sub_script
```

[98]:
```
from imp import reload
reload(MyMainPackage.MySubPackage.sub_script)
```

```
I am inside sub_script, which is located in MySubPackage
The value of the variable X is: 23
```

[98]:
```
<module 'MyMainPackage.MySubPackage.sub_script' from 'C:\\Users\\vakanski\\Documents\\
↪Codes\\2023 Codes\\Python for Data Science Course\\Lectures_2023\\Theme_1-Python_
↪Programming\\Lecture_6-Exceptions,_Modules\\Posted\\Lecture_6-Exceptions,_Modules\\
↪MyMainPackage\\MySubPackage\\sub_script.py'>
```

Once imported, `sub-script` becomes a module object nested in the object `MySubPackage`, which in turn is nested in the object `MyMainPackage`.

Similarly, `MySubPackage` is a module object that is nested in the object `MyMainPackage`.

[99]:
```
MyMainPackage.MySubPackage
```

[99]:
```
<module 'MyMainPackage.MySubPackage' from 'C:\\Users\\vakanski\\Documents\\Codes\\2023␣
↪Codes\\Python for Data Science Course\\Lectures_2023\\Theme_1-Python_Programming\\
↪Lecture_6-Exceptions,_Modules\\Posted\\Lecture_6-Exceptions,_Modules\\MyMainPackage\\
↪MySubPackage\\__init__.py'>
```

Python Path

If the directory `MyMainPackage` is not in the current working directory, then it may need to be added to the Python search path. To do that, either add the full path to the directory to the PYTHONPATH variable (by setting the Environment Variables on Windows systems), or the path to the directory can be added to a `.pth` file. Note that if the package is a standard library directory of a built-in function (e.g., `random`, `time`, `sys`, `os`), or if it is located in the site-packages directory (where third-party libraries are installed), it will be automatically found by Python, and it does not need to be added to the Python search path.

Alternatively, the path to the directory can be manually added using `sys.path` (that is, the `path` attribute of the standard library module `sys`). For instance, I can examine the `sys.path` on my computer, as shown in the following cell. Since the `sys.path` is just a list of directories, we can manually add the path of the current working directory, by using the `append` to list method.

[100]:
```
import sys
sys.path
```

[100]:
```
['C:\\Users\\vakanski\\Documents\\Codes\\2023 Codes\\Python for Data Science Course\\
↪Lectures_2023\\Theme_1-Python_Programming\\Lecture_6-Exceptions,_Modules\\Posted\\
↪Lecture_6-Exceptions,_Modules',
 'C:\\Users\\vakanski\\anaconda3\\python311.zip',
 'C:\\Users\\vakanski\\anaconda3\\DLLs',
 'C:\\Users\\vakanski\\anaconda3\\Lib',
 'C:\\Users\\vakanski\\anaconda3',
 '',
 'C:\\Users\\vakanski\\anaconda3\\Lib\\site-packages',
 'C:\\Users\\vakanski\\anaconda3\\Lib\\site-packages\\win32',
 'C:\\Users\\vakanski\\anaconda3\\Lib\\site-packages\\win32\\lib',
 'C:\\Users\\vakanski\\anaconda3\\Lib\\site-packages\\Pythonwin']
```

[101]:
```
sys.path.append('C:\\Users\\Alex\\Desktop\\python\\Lecture 6 Module Packages')
```

[102]:
```
sys.path
# The appended path is listed last
```

[102]:
```
['C:\\Users\\vakanski\\Documents\\Codes\\2023 Codes\\Python for Data Science Course\\
↪Lectures_2023\\Theme_1-Python_Programming\\Lecture_6-Exceptions,_Modules\\Posted\\
↪Lecture_6-Exceptions,_Modules',
 'C:\\Users\\vakanski\\anaconda3\\python311.zip',
 'C:\\Users\\vakanski\\anaconda3\\DLLs',
 'C:\\Users\\vakanski\\anaconda3\\Lib',
 'C:\\Users\\vakanski\\anaconda3',
 '',
 'C:\\Users\\vakanski\\anaconda3\\Lib\\site-packages',
 'C:\\Users\\vakanski\\anaconda3\\Lib\\site-packages\\win32',
 'C:\\Users\\vakanski\\anaconda3\\Lib\\site-packages\\win32\\lib',
 'C:\\Users\\vakanski\\anaconda3\\Lib\\site-packages\\Pythonwin',
 'C:\\Users\\Alex\\Desktop\\python\\Lecture 6 Module Packages']
```

Notice now that the directory `MyMainPackage` is now listed in the `sys.path`. However, this modified `sys.path` is temporary and it is valid only for the duration of the current session; the path is refreshed every time Jupyter Notebook is restarted, or the notebook kernel is shut down. On the other hand, the path configuration in `PYTHONPATH` is permanent, and it lives after the current session is terminated.

Package Relative Imports

To illustrate package relative imports in Python we will use the `MyRelativeImportPackage` which is similar to the `MyMainPackage` and contains several simple files.

```
MyRelativeImportPackage
    ├── __init__.py
    ├── relative_import_script_1
    ├── relative_import_script_2
    ├── relative_import_script_5
    ├── relative_import_script_6
    ├── script_1
    ├── script_2
    ├── script_3
    ├── script_4
    ├── MySubPackage
    │   ├── __init__.py
    │   ├── relative_import_script_3
    │   ├── relative_import_script_4
    │   ├── sub_script
```

When modules within a package need to import other names from other modules in the same package, it is still possible to use the full path syntax for importing, as we did in the above section. This is called an **absolute import**.

For instance, the `relative_import_script_1.py` in the first line imports `script_1` by using the full name of the directory (i.e., `from MyRelativeImportPackage import script_1`).

relative_import_script_1.py

```
1  from MyRelativeImportPackage import script_1
2
3  print(20*'-')
4  print('I am inside the relative_import_scipt_1, which is located in MyMainPackage')
```

script_1.py

```
1  print('I am inside script_1, which is located in MyMainPackage')
2
```

[103]:
```
import MyRelativeImportPackage.relative_import_script_1
```

```
I am inside script_1, which is located in MyMainPackage
--------------------
I am inside the relative_import_scipt_1, which is located in MyMainPackage
```

However, package files can also make use of a special syntax to simplify import statements within the same package. Instead of directly using the full path to the directory, Python allows to use a leading dot `.` to refer to the current directory in the package.

Therefore, instead of using `from MyRelativeImportPackage import script_1`, we can use `from . import script_1`. This is implemented in the `relative_import_script_2.py` to import `script_2`.

This syntax is referred to as a **relative import** because the path to the module to be imported is related to the current directory in which the module that imports is located.

The convenience of using relative imports is that we don't need to write the name or the path of the current directory.

relative_import_script_2.py

```
1  from . import script_2
2
3  print(20*'-')
4  print('I am inside the relative_import_scipt_2, which is located in MyMainPackage')
```

script_2.py

```
1  print('I am inside script_2, which is located in MyMainPackage')
2
```

[104]:
```
import MyRelativeImportPackage.relative_import_script_2
```

```
I am inside script_2, which is located in MyMainPackage
--------------------
I am inside the relative_import_scipt_2, which is located in MyMainPackage
```

One more example is presented in the next cell, where the module `relative_import_script_3.py` is located in the directory `MySubPackage` and it imports the module `sub_script` which is located in the same directory by using the `.` syntax.

relative_import_script_3.py

```
1  from . import sub_script
2
3  print(20*'-')
4  print('I am inside the relative_import_scipt_3, which is located in MySubPackage')
```

sub_script.py

```
1  print('I am inside sub_script, which is located in MySubPackage')
2
3
4
```

[105]:
```
import MyRelativeImportPackage.MySubPackage.relative_import_script_3
```

```
I am inside sub_script, which is located in MySubPackage
--------------------
I am inside the relative_import_scipt_3, which is located in MySubPackage
```

If we use two dots syntax as in `..`, then a module can import another module that is located in its parent directory of the current package (i.e., the directory above). For example, the `relative_import_script_4.py` is located in the

MySubPackage directory, and it uses `from .. import script_3` to import the `script_3` module that is located in the parent directory of `MySubPackage`, that is, `MyMainPacakage`.

relative_import_script_4.py

```
1  from .. import script_3
2
3  print(20*'-')
4  print('I am inside the relative_import_scipt_4, which is located in MySubPackage')
```

script_3.py

```
1  print('I am inside script_3, which is located in MyMainPackage')
2
```

[106]:
```
import MyRelativeImportPackage.MySubPackage.relative_import_script_4
```

```
I am inside script_3, which is located in MyMainPackage
--------------------
I am inside the relative_import_scipt_4, which is located in MySubPackage
```

On the other hand, if we tried to use only `import script_3` instead of `from . import script_3`, this will fail. We must use the `from` dotted syntax to import modules located in the same package. This is illustrated in the example in the following cell.

relative_import_script_5.py

```
1  import script_3
2
3  print(20*'-')
4  print('I am inside the relative_import_scipt_5, which is located in MyMainPackage')
```

[107]:
```
import MyRelativeImportPackage.relative_import_script_5
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[107], line 1
----> 1 import MyRelativeImportPackage.relative_import_script_5

File ~\Documents\Codes\2023 Codes\Python for Data Science Course\Lectures_2023\Theme_1-
↪Python_Programming\Lecture_6-Exceptions,_Modules\Posted\Lecture_6-Exceptions,_Modules\
↪MyRelativeImportPackage\relative_import_script_5.py:1
----> 1 import script_3
      3 print(20*'-')
      4 print('I am inside the relative_import_scipt_5, which is located in MyMainPackage
↪')

ModuleNotFoundError: No module named 'script_3'
```

Another way to use the relative imports is shown in the `relative_import_script_6.py` module, where the syntax

`from .script_4 import X` is used to import the name `X` from the `script_4` module which is located in the same directory as the importer module. This way, we can import specific names from modules in the same package.

relative_import_script_6.py

```
1  from .script_4 import X
2
3  print(20*'-')
4  print('I am inside the relative_import_scipt_6, which is located in MyMainPackage')
5  print('The value of the variable X is', X)
```

script_4.py

```
1  print('I am inside script_4, which is located in MyMainPackage')
2
3  X = 5
4
5
```

[108]:
```
import MyRelativeImportPackage.relative_import_script_6
```

```
I am inside script_4, which is located in MyMainPackage
--------------------
I am inside the relative_import_scipt_6, which is located in MyMainPackage
The value of the variable X is 5
```

Absolute imports are often preferred because they are straightforward, and it is easy to tell exactly where the imported module or name is located, just by looking at the statement. But, they require more typing and writing full names and paths in the code.

One clear advantage of relative imports is that they are quite succinct, and they can turn a very long import statement into a simple and short statement. Relative imports can be messy, particularly for projects where the organization of the directories is likely to change. Relative imports are also not as readable as absolute ones, and it is not easy to tell the location of the imported names.

7.6.5 References

1. Mark Lutz, "Learning Python," 5-th edition, O-Reilly, 2013. ISBN: 978-1-449-35573-9.
2. Pierian Data Inc., "Complete Python 3 Bootcamp," codes available at: https://github.com/Pierian-Data/Complete-Python-3-Bootcamp.

BACK TO TOP

7.7 Lecture 7 - NumPy for Array Operations

7.7.1 7.1 Introduction to NumPy

NumPy (short for Numerical Python) was created in 2005, and since then, the NumPy library has evolved into an essential library for scientific computing in Python. It has become a building block of many other scientific libraries, such as SciPy, Scikit-learn, Pandas, and others.

NumPy provides a convenient Python interface for working with multi-dimensional array data structures. The NumPy array data structure is also called `ndarray`, which is short for *n*-dimensional array.

In addition to being mostly implemented in C and using Python as a "glue language," the main reason why NumPy is so efficient for numerical computations is that NumPy arrays use contiguous blocks of memory that can be efficiently cached by the CPU. In contrast, Python lists are arrays of pointers to objects in random locations in memory, which cannot be easily cached and come with a more expensive memory look-up. On the other hand, NumPy arrays have a fixed size and are homogeneous, which means that all elements in an array must have the same type. Homogenous `ndarray` objects have the advantage that NumPy can carry out operations using efficient C code and avoid expensive type checks and other overheads of the Python API.

N-dimensional Arrays

NumPy is built around `ndarrays <https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html>`__ objects, which are multi-dimensional array data structures. Intuitively, we can think of a one-dimensional NumPy array as a vector of elements – you may think of it as a fixed-size Python list where all elements share the same type. Similarly, we can think of a two-dimensional array as a data structure to represent a matrix (or a Python list of lists). NumPy arrays can have up to 32 dimensions. For instance, RGB images have 3 dimensions, corresponding to pixels width and height, and the three color channels. Similarly, an RGB video has 4 dimensions, corresponding to pixels width and height, color channel, and frame number.

In this next example, we will call the `array` function first to create an one-dimensional NumPy array, and afterward, a two-dimensional NumPy array, consisting of two rows and three columns (from a list of lists).

[1]:
```
import numpy as np

ary1d = [1., 2., 3.]
np.array(ary1d)
```

[1]:
```
array([1., 2., 3.])
```

[2]:
```
lst = [[1, 2, 3],
       [4, 5, 6]]

ary2d = np.array(lst)
ary2d

# rows x columns
```

[2]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

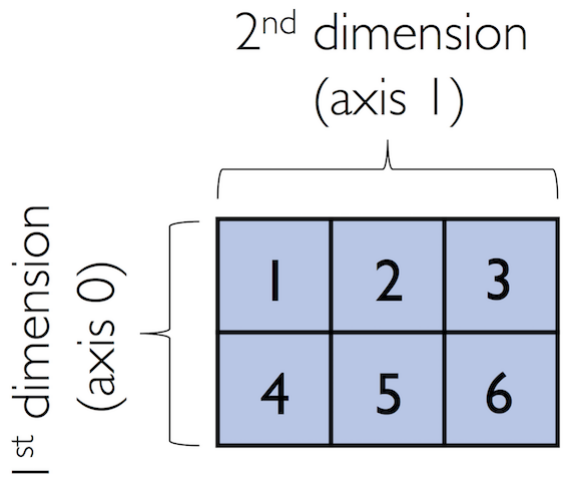


Figure source: Reference [1].

By default, NumPy infers the type of the array upon construction. Since we passed Python integers to the array, the `ndarray` object `ary2d` should be of type `int32`, which we can confirm by accessing the `dtype` attribute.

[3]:
```
ary2d.dtype
```

[3]:
```
dtype('int32')
```

If we want to construct NumPy arrays of different types, we can pass an argument for the `dtype` parameter of the array using the `astype` method (for example `np.int64` to create 64-bit arrays). For a full list of supported data types, please refer to the official NumPy documentation.

[4]:
```
int64_ary = ary2d.astype(np.int64)
int64_ary
```

[4]:
```
array([[1, 2, 3],
       [4, 5, 6]], dtype=int64)
```

[5]:
```
float32_ary = ary2d.astype(np.float32)
float32_ary
```

[5]:
```
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
```

[6]:
```
float32_ary.dtype
```

[6]:
```
dtype('float32')
```

To return the number of elements in an array, we can use the `size` attribute, as shown below.

[7]:
```
ary2d.size
```

[7]:
```
6
```

And the number of dimensions of our array can be obtained via the `ndim` attribute.

[8]:
```
ary2d.ndim
```

[8]:
```
2
```

If we are interested in the number of elements along each array dimension (in the context of NumPy arrays, we may also refer to them as *axes*), we can access the `shape` attribute as shown below.

[9]:
```
ary2d.shape
```

[9]:
```
(2, 3)
```

The `shape` is always a tuple; in the code example above, the two-dimensional `ary2d` object has two *rows* and *three* columns, `(2, 3)`.

The `shape` of a one-dimensional array only contains a single value.

[10]:
```
np.array([1., 2., 3.]).shape
```

[10]:
```
(3,)
```

7.7.2 7.2 Array Construction and Indexing

This section provides several useful functions to construct arrays, e.g., containing only ones or zeros.

[11]:
```
np.ones((3, 4), dtype=int)
```

[11]:
```
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
```

[12]:
```
np.zeros((3, 3))
```

[12]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

We can use these functions to create arrays with arbitrary values, e.g., we can create an array containing the values 99 as follows.

[13]:
```
np.zeros((3, 3)) + 99
```

[13]:
```
array([[99., 99., 99.],
       [99., 99., 99.],
       [99., 99., 99.]])
```

Creating arrays of ones or zeros can also be useful as placeholder arrays, in cases where we do not want to use the initial values for computations right away.

NumPy also has functions to create identity matrices and diagonal matrices as `ndarrays`.

[14]:
```
np.eye(3)
```

[14]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

[15]:
```
np.diag((1, 2, 3))
```

[15]:
```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

Two other very useful functions for creating sequences of numbers within a specified range are `arange` and `linspace`. NumPy's `arange` function follows the same syntax as Python's `range` function. If two arguments are provided, the first argument represents the start value and the second argument defines the stop value of a half-open interval.

[16]:
```
np.arange(4, 10)
```

[16]:
```
array([4, 5, 6, 7, 8, 9])
```

If we only provide a single argument, the default start value of 0 is assumed.

[17]:
```
np.arange(5)
```

[17]:
```
array([0, 1, 2, 3, 4])
```

Similar to Python's `range`, a third argument can be provided to define the *step* (the default step size is 1). For example, we can obtain an array of all values between 1 and 11 with 0.1 step as follows.

[18]:
```
ary3 = np.arange(1., 11., 0.1)
ary3
```

[18]:
```
array([ 1. ,  1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9,  2. ,
        2.1,  2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9,  3. ,  3.1,
        3.2,  3.3,  3.4,  3.5,  3.6,  3.7,  3.8,  3.9,  4. ,  4.1,  4.2,
        4.3,  4.4,  4.5,  4.6,  4.7,  4.8,  4.9,  5. ,  5.1,  5.2,  5.3,
        5.4,  5.5,  5.6,  5.7,  5.8,  5.9,  6. ,  6.1,  6.2,  6.3,  6.4,
        6.5,  6.6,  6.7,  6.8,  6.9,  7. ,  7.1,  7.2,  7.3,  7.4,  7.5,
        7.6,  7.7,  7.8,  7.9,  8. ,  8.1,  8.2,  8.3,  8.4,  8.5,  8.6,
        8.7,  8.8,  8.9,  9. ,  9.1,  9.2,  9.3,  9.4,  9.5,  9.6,  9.7,
        9.8,  9.9, 10. , 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8,
       10.9])
```

Note the shape of this array.

[19]:
```
np.shape(ary3)
```

[19]:
```
(100,)
```

The `linspace` function is especially useful if we want to create a number of evenly spaced values in a specified interval.

[20]:
```
np.linspace(6., 25., num=5)
```

[20]:
```
array([ 6.  , 10.75, 15.5 , 20.25, 25.  ])
```

Array Indexing

Simple NumPy indexing and slicing work similar to Python lists, as in the following examples.

[21]:
```
ary = np.array([1, 2, 3, 4])
ary[2]
```

[21]:
```
3
```

Also, the same Python semantics apply to slicing operations. The following example shows how to fetch the first three elements in `ary`.

[22]:
```
ary[0:3]
```

[22]:
```
array([1, 2, 3])
```

If we work with arrays that have more than one dimension or axis, we separate our indexing or slicing operations by commas as shown in the following examples.

[23]:
```
ary = np.array([[1, 2, 3],
                [4, 5, 6]])

ary[0, -2] # first row, second from last element
```

[23]:
```
2
```

[24]:
```
ary[-1, -1] # lower right
```

[24]:
```
6
```

[25]:
```
ary[1, 1] # first row, second column
```

[25]:
```
5
```

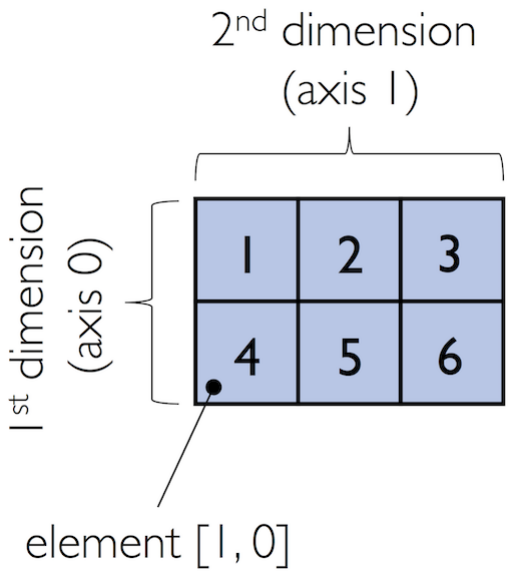


Figure source: Reference [1].

[26]:
```
ary[:, 0] # entire first column
```

[26]:
```
array([1, 4])
```

[27]:
```
ary[:, :2] # first two columns
```

[27]:
```
array([[1, 2],
       [4, 5]])
```

7.7.3 7.3 Array Math

One of the core features of NumPy that makes working with `ndarray` so efficient and convenient is **vectorization**. NumPy provides vectorized functions for performing element-wise operations implicitly via so-called *ufuncs* which is short for "universal functions".

There are more than 60 ufuncs available in NumPy; ufuncs are implemented in compiled C code and very fast and efficient compared to Python.

To provide an example of a simple ufunc for element-wise addition, consider the following example, where we add a scalar 1 to each element in a nested Python list.

[28]:
```
lst = [[1, 2, 3],
       [4, 5, 6]] # 2d array

for row_idx, row_val in enumerate(lst):
    for col_idx, col_val in enumerate(row_val):
        lst[row_idx][col_idx] += 1
lst
```

[28]:
```
[[2, 3, 4], [5, 6, 7]]
```

We can accomplish the same using NumPy's ufunc for element-wise scalar addition as shown below.

[29]:
```
ary = np.array([[1, 2, 3], [4, 5, 6]])
ary = np.add(ary, 1)
ary
```

[29]:
```
array([[2, 3, 4],
       [5, 6, 7]])
```

For basic arithmetic operations we can use `add`, `subtract`, `divide`, `multiply`, `power`, and `exp` (exponential). However, NumPy uses operator overloading so that we can use mathematical operators (+, -, /, *, and **) directly.

[30]:
```
np.add(ary, 1)
```

[30]:
```
array([[3, 4, 5],
       [6, 7, 8]])
```

[31]:
```
ary + 1
```

[31]:
```
array([[3, 4, 5],
       [6, 7, 8]])
```

[32]:
```
np.power(ary, 2)
```

[32]:
```
array([[ 4,  9, 16],
       [25, 36, 49]], dtype=int32)
```

[33]:
```
ary**2
```

[33]:
```
array([[ 4,  9, 16],
       [25, 36, 49]])
```

NumPy also has implementations for other math operations, such as `sqrt` (square root), `log` (natural logarithm), and `log10` (base-10 logarithm).

[34]:
```
np.sqrt(ary)
```

[34]:
```
array([[1.41421356, 1.73205081, 2.        ],
       [2.23606798, 2.44948974, 2.64575131]])
```

Often, we want to compute the sum or product of array elements along a given axis. For this purpose, we can use the `reduce` operation. By default, `reduce` applies an operation along the first axis (`axis=0`). In the case of a two-dimensional array, we can think of the first axis as the rows of a matrix. Thus, adding up elements along rows yields the column sums of that matrix as shown below.

[35]:
```
ary = np.array([[1, 2, 3],
                [4, 5, 6]]) # rolling over the 1st axis, axis 0

np.add.reduce(ary, axis=0)
```

[35]:
```
array([5, 7, 9])
```

To compute the row sums of the array above, we can specify `axis=1`.

[36]:
```
np.add.reduce(ary, axis=1) # row sums
```

[36]:
```
array([ 6, 15])
```

NumPy also provides functions for specific operations such as `product` and `sum`. For example, `sum(axis=0)` is equivalent to `add.reduce`.

[37]:
```
ary.sum(axis=0) # column sums
```

[37]:
```
array([5, 7, 9])
```

[38]:
```
ary.sum(axis=1) # row sums
```

[38]:
```
array([ 6, 15])
```

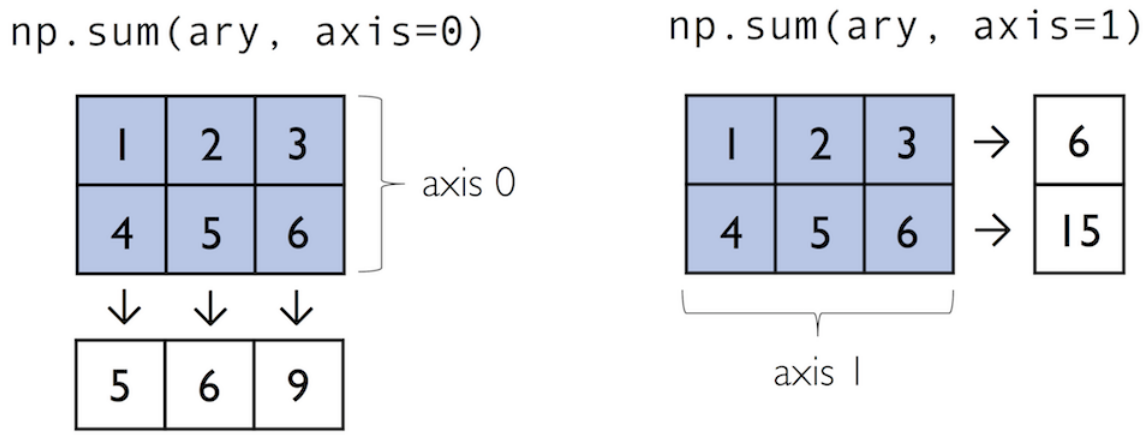


Figure source: Reference [1].

Note also that `product` and `sum` both compute the product or sum of the entire array if we do not specify an axis.

[39]:
```
ary.sum()
```

[39]:
```
21
```

Other useful ufuncs are:

- `np.mean` (computes arithmetic mean or average)
- `np.std` (computes the standard deviation)
- `np.var` (computes variance)
- `np.sort` (sorts an array)
- `np.argsort` (returns indices that would sort an array)
- `np.min` (returns the minimum value of an array)
- `np.max` (returns the maximum value of an array)
- `np.argmin` (returns the index of the minimum value)
- `np.argmax` (returns the index of the maximum value)
- `np.array_equal` (checks if two arrays have the same shape and elements)

7.7.4 7.4 Broadcasting

Broadcasting allows to perform vectorized operations between two arrays even if their dimensions do not match.

[40]:
```
ary1 = np.array([1, 2, 3])
ary1 + 1
```

[40]:
```
array([2, 3, 4])
```

[41]:
```
# this is equivalent to:
ary1 + np.array([1, 1, 1])
```

[41]:
```
array([2, 3, 4])
```

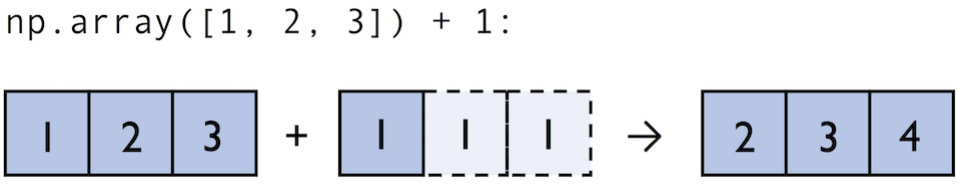


Figure source: Reference [1].

For example, we can add a one-dimensional to a two-dimensional array, where NumPy creates an implicit multidimensional grid from the one-dimensional array `ary1`.

[42]:
```
ary2 = np.array([[4, 5, 6],
                 [7, 8, 9]])

ary2 + ary1
```

[42]:
```
array([[ 5,  7,  9],
       [ 8, 10, 12]])
```

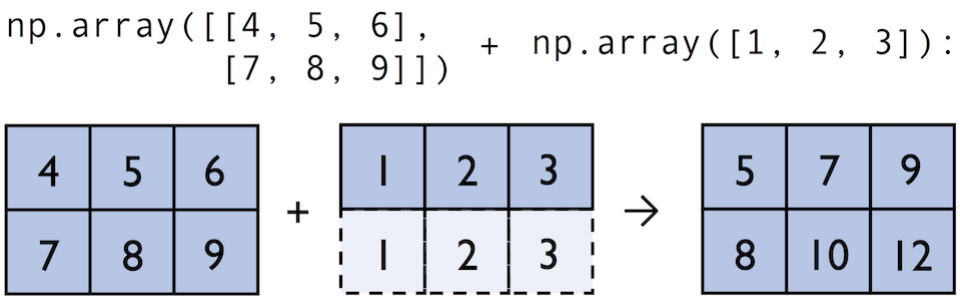


Figure source: Reference [1].

Using broadcasting in Python requires compatibility between the sizes of the arrays. For instance, if we try to add two arrays of sizes (3,3) and (2,2), Python will return an error.

[43]:
```
np.ones((3,3)) + np.ones((2,2))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[43], line 1
----> 1 np.ones((3,3)) + np.ones((2,2))

ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

Advanced Indexing – Memory Views and Copies

In the previous sections, we used basic indexing and slicing routines. It is important to note that basic integer-based indexing and slicing create so-called *views* of NumPy arrays in the memory. Working with views can be highly desirable since it avoids making unnecessary copies of arrays to save memory resources. To illustrate the concept of memory views, let's look at a simple example where we access the first row in an array, assign it to a variable, and modify that variable.

[44]:
```
ary = np.array([[1, 2, 3],
                [4, 5, 6]])

first_row = ary[0, :]
```

[45]:
```
first_row
```

[45]:
```
array([1, 2, 3])
```

[46]:
```
first_row += 99
```

[47]:
```
first_row
```

[47]:
```
array([100, 101, 102])
```

As expected, `first_row` was modified, now containing the original values in the first row incremented by 99.

Note, however, that the original array was modified as well. The reason for this is that `ary[0, :]` created a view of the first row in `ary`, and its elements were then incremented by 99.

[48]:
```
ary
```

[48]:
```
array([[100, 101, 102],
       [  4,   5,   6]])
```

The same concept applies to slicing operations.

[49]:
```
ary = np.array([[1, 2, 3],
                [4, 5, 6]])

last_col = ary[:, 2]
last_col += 99
ary
```

[49]:
```
array([[  1,   2, 102],
       [  4,   5, 105]])
```

Therefore, it is important to be aware that indexing and slicing create views, which can speed up our code by avoiding to create unnecessary copies in memory. However, in certain scenarios we want to create a copy of an array; we can do this via the `copy` method as shown below.

[50]:
```
ary = np.array([[1, 2, 3],
                [4, 5, 6]])

first_row = ary[0].copy()
first_row += 99
```

[51]:
```
first_row
```

[51]:
```
array([100, 101, 102])
```

[52]:
```
ary
```

[52]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

Fancy Indexing

In addition to basic single-integer indexing and slicing operations, NumPy supports advanced indexing routines called *fancy indexing*. Via fancy indexing, we can use tuple or list objects of non-contiguous integer indices to return desired array elements. Since fancy indexing can be performed with non-contiguous sequences, it cannot return a view – a contiguous slice from memory. Thus, fancy indexing always returns a copy of an array: it is important to keep that in mind. The following code snippets show some fancy indexing examples.

[53]:
```
ary = np.array([[1, 2, 3],
                [4, 5, 6]])

ary[:, [0, 2]] # first and and last column
```

[53]:
```
array([[1, 3],
       [4, 6]])
```

[54]:
```
this_is_a_copy = ary[:, [0, 2]]
this_is_a_copy += 99
```

Note that the values in `this_is_a_copy` were incremented as expected.

[55]:
```
this_is_a_copy
```

[55]:
```
array([[100, 102],
       [103, 105]])
```

However, the contents of the original array remain unaffected.

[56]:
```
ary
```

[56]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

Boolean Masks for Indexing

We can also use Boolean masks for indexing, that is, arrays of `True` and `False` values. Consider the following example, where we return all values in the array that are greater than 3.

[57]:
```
ary = np.array([[1, 2, 3],
                [4, 5, 6]])

greater3_mask = ary > 3
greater3_mask
```

[57]:
```
array([[False, False, False],
       [ True,  True,  True]])
```

Using these masks, we can select elements given our desired criteria.

[58]:
```
ary[greater3_mask]
```

[58]:
```
array([4, 5, 6])
```

Or, we can also write this as follows.

[59]:
```
ary[ary>3]
```

[59]:
```
array([4, 5, 6])
```

We can also chain different selection criteria using the logical *and* operator `&` or the logical *or* operator `|`. The example below demonstrates how we can select array elements that are greater than 3 and divisible by 2.

[60]:
```
(ary > 3) & (ary % 2 == 0)
```

[60]:
```
array([[False, False, False],
       [ True, False,  True]])
```

Similar to the previous example, we can use this boolean array as a mask for selecting the respective elements from the array.

[61]:
```
ary[(ary > 3) & (ary % 2 == 0)]
```

[61]:
```
array([4, 6])
```

And, for example, to negate a condition, we can use the ~ operator:

[62]:
```
ary[~((ary < 2) | (ary > 4) )]
```

[62]:
```
array([2, 3, 4])
```

A related, useful function to assign values to specific elements in an array is the `np.where` function. In the example below, we assign a 1 to all values in the array that are greater than 2, and 0 otherwise.

[63]:
```
ary = np.array([1, 2, 3, 4, 5])

np.where(ary > 2, 1, 0)
```

[63]:
```
array([0, 0, 1, 1, 1])
```

7.7.5 7.5 Random Number Generators

In machine learning and data science, we often need to generate arrays of random numbers, such as the initial values of the model parameters. NumPy has a `random` subpackage to create random numbers and samples from a variety of distributions.

Let's start with drawing a random sample of three numbers in the range [0,1] from a uniform distribution via `random.rand`.

[64]:
```
np.random.rand(3)
```

[64]:
```
array([0.620625  , 0.58006405, 0.4566714 ])
```

If we make another draw, we will obtain a different random sample.

[65]:
```
np.random.rand(3)
```

[65]:
```
array([0.93181132, 0.1938188 , 0.85219216])
```

Numpy also allows to use a fixed *random seed* for the random number generator, and in that case, the random sample will be same at each draw.

[66]:
```
np.random.seed(seed=123)

np.random.rand(3)
```

[66]:
```
array([0.69646919, 0.28613933, 0.22685145])
```

[67]:
```
np.random.seed(seed=123)

np.random.rand(3)
```

[67]:
```
array([0.69646919, 0.28613933, 0.22685145])
```

And, of course, if we change the random seed value, the generated random numbers will be also different.

[68]:
```
np.random.seed(seed=43)

np.random.rand(3)
```

[68]:
```
array([0.11505457, 0.60906654, 0.13339096])
```

Using a random seed is highly recommended in practical applications and in research projects, since it ensures that our results are reproducible, since using the same seed will create the same random numbers.

We can also create multi-dimensional arrays of random numbers, if needed.

[69]:
```
np.random.rand(5,5)
```

[69]:
```
array([[0.24058962, 0.32713906, 0.85913749, 0.66609021, 0.54116221],
       [0.02901382, 0.7337483 , 0.39495002, 0.80204712, 0.25442113],
       [0.05688494, 0.86664864, 0.221029  , 0.40498945, 0.31609647],
       [0.0766627 , 0.84322469, 0.84893915, 0.97146509, 0.38537691],
       [0.95448813, 0.44575836, 0.66972465, 0.08250005, 0.89709858]])
```

And, we can draw random numbers from other distributions. For instance, `random.randn` returns random numbers from a normal, i.e., Gaussian, distribution, with mean 0, and variance 1.

[70]:
```
np.random.randn(4)
```

[70]:
```
array([-1.04683899, -0.88961759,  0.01404054, -0.16082969])
```

Or, for instance if we needed to generate 6 random values from a normal distribution with mean -3 and standard deviation 10, we can write as follows.

[71]:
```
-3 + 10*np.random.randn(6)
```

[71]:
```
array([19.30359649, -6.99115719, -2.45555437,  5.84181815, -4.07980561,
        2.55606984])
```

Also, it can be useful to create separate `RandomState` objects for various parts of our code, so that we can test methods in a reproducible manner. The example below shows how we can use a `RandomState` object to create the same results that we obtained via `np.random.rand` in the previous code.

[72]:
```
rng2 = np.random.RandomState(seed=123)
rng2.rand(3)
```

[72]:
```
array([0.69646919, 0.28613933, 0.22685145])
```

In the above code, `np.random.RandomState()` constructed a random number generator that we named `rng2`. When we call `rng2` to draw random numbers, it uses the specified seed (seed=123) to create random numbers in a controlled way. Note however that the seed 123 will be applied only to the `rng2` object, and it does not have effect on other freestanding functions in our code that use `np.random`. If we would like to apply the random seed 123 to all other `np.random` functions in our code, we will need to explicitly request this with `np.random.seed(seed=123)`.

7.7.6 7.6 Reshaping NumPy Arrays

In practice, we often run into situations where existing arrays do not have the *right* shape to perform certain computations. As you might remember from the beginning of this article, the size of NumPy arrays is fixed. Fortunately, this does not mean that we have to create new arrays and copy values from the old array to the new one if we want arrays of different shapes. That is, the size is fixed, but the shape is not. NumPy provides a `reshape` method that allows to obtain a view of an array with a different shape.

For example, we can reshape a one-dimensional array into a two-dimensional one using `reshape` as follows.

[73]:
```
ary1d = np.array([1, 2, 3, 4, 5, 6])
ary2d_view = ary1d.reshape(2, 3)
ary2d_view
```

[73]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

[74]:
```
np.may_share_memory(ary2d_view, ary1d)
```

[74]:
```
True
```

The `True` value returned from `np.may_share_memory` indicates that the reshape operation returns a memory view, not a copy.

When using reshape, we need to make sure that the reshaped array has the same number of elements as the original one. Otherwise, we will obtain an error message.

[75]:
```
ary2d_view1 = ary1d.reshape(3, 4)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[75], line 1
----> 1 ary2d_view1 = ary1d.reshape(3, 4)

ValueError: cannot reshape array of size 6 into shape (3,4)
```

However, we do not need to specify the number of elements in each axis. NumPy can figure out how many elements to put along an axis if only one axis is unspecified, by using the placeholder `-1`.

[76]:
```
ary1d.reshape(-1, 2)
```

[76]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

We can also use `reshape` to flatten an array.

[77]:
```
ary = np.array([[[1, 2, 3],
                 [4, 5, 6]]])

ary.reshape(-1)
```

[77]:
```
array([1, 2, 3, 4, 5, 6])
```

Other methods for flattening arrays exist, namely `flatten`, which creates a copy of the array, and `ravel`, which creates a memory view.

[78]:
```
ary.flatten()
```

[78]:
```
array([1, 2, 3, 4, 5, 6])
```

[79]:
```
ary.ravel()
```

[79]:
```
array([1, 2, 3, 4, 5, 6])
```

Sometimes, we are interested in merging different arrays. Unfortunately, there is no efficient way to do this without creating a new array, since NumPy arrays have a fixed size. While combining arrays should be avoided if possible for reasons of computational efficiency, it is sometimes necessary. To combine two or more array objects, we can use NumPy's `concatenate` function as shown in the following examples.

[80]:
```
ary = np.array([[1, 2, 3]])

# stack along the first axis (here: rows)
np.concatenate((ary, ary), axis=0)
```

[80]:
```
array([[1, 2, 3],
       [1, 2, 3]])
```

[81]:
```
# stack along the second axis (here: columns)
np.concatenate((ary, ary), axis=1)
```

[81]:
```
array([[1, 2, 3, 1, 2, 3]])
```

Two related functions are `vstack` and `hstack` that stand for vertically or horizontally stacking arrays, respectively.

[82]:
```
np.vstack((ary, ary))
```

[82]:
```
array([[1, 2, 3],
       [1, 2, 3]])
```

[83]:
```
np.hstack((ary, ary))
```

[83]:
```
array([[1, 2, 3, 1, 2, 3]])
```

7.7.7 7.7 Linear Algebra with NumPy

We can think of one-dimensional NumPy arrays as data structures that represent row vectors.

[84]:
```
row_vector = np.array([1, 2, 3])
row_vector
```

[84]:
```
array([1, 2, 3])
```

[85]:
```
row_vector.shape
```

[85]:
```
(3,)
```

Similarly, we can use two-dimensional arrays to create column vectors.

[86]:
```
column_vector = np.array([1, 2, 3]).reshape(-1, 1)
column_vector
```

[86]:
```
array([[1],
       [2],
       [3]])
```

[87]:
```
column_vector.shape
```

[87]:
```
(3, 1)
```

Instead of reshaping a one-dimensional array into a two-dimensional one, we can simply add a new axis as shown below.

[88]:
```
row_vector[:, np.newaxis]
```

[88]:
```
array([[1],
       [2],
       [3]])
```

To perform matrix multiplication between matrices, we know that number of columns of the left matrix must match the number of rows of the matrix to the right. In NumPy, we can perform matrix multiplication via the `matmul` function.

[89]:
```
matrix = np.array([[1, 2, 3],
                   [4, 5, 6]])
```

[90]:
```
np.matmul(matrix, column_vector)
```

[90]:
```
array([[14],
       [32]])
```

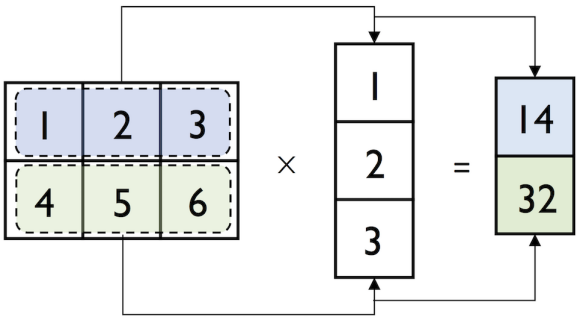


Figure source: Reference [1].

However, if we are working with matrices and vectors, NumPy can be quite forgiving if the dimensions of matrices and one-dimensional arrays do not match exactly – thanks to broadcasting. The following example yields the same result as the matrix-column vector multiplication, except that it returns a one-dimensional array instead of a two-dimensional one.

[91]:
```
np.matmul(matrix, row_vector)
```

[91]:
```
array([14, 32])
```

Similarly, we can compute the dot-product between two vectors.

[92]:
```
np.matmul(row_vector, row_vector)
```

[92]:
```
14
```

NumPy has a special `dot` function that calculates the dot-product for one-dimensional arrays, and otherwise behaves similar to `matmul` on multi-dimensional arrays.

[93]:
```
np.dot(matrix, row_vector)
```

[93]:
```
array([14, 32])
```

Note that an even more convenient way for executing `np.dot` is using the `@` symbol with NumPy arrays.

[94]:
```
matrix @ row_vector
```

[94]:
```
array([14, 32])
```

NumPy also has a handy `transpose` method to transpose matrices.

[95]:
```
matrix = np.array([[1, 2, 3],
                   [4, 5, 6]])

matrix.transpose()
```

[95]:
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

There is also a shorthand for `transpose` simply as `T`.

[96]:
```
matrix.T
```

[96]:
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

While this section demonstrates some of the basic linear algebra operations carried out on NumPy arrays that we use in practice, you can find additional functions in the documentation of NumPy's submodule for linear algebra: `numpy.linalg <https://docs.scipy.org/doc/numpy/reference/routines.linalg.html>`__. If you want to perform a particular linear algebra routine that is not implemented in NumPy, it is also worth consulting the `scipy.linalg` documentation <https://docs.scipy.org/doc/scipy/reference/linalg.html>`__ – SciPy is a library for scientific computing built on top of NumPy.

One last note is that there is also a special `matrix <https://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html>`__ type in NumPy. NumPy `matrix` objects are analogous to NumPy arrays but are restricted to two dimensions. Also, matrices define certain operations differently than arrays; for instance, the `*` operator performs matrix multiplication instead of element-wise multiplication. However, NumPy `matrix` is less popular in the science community compared to the more general array data structure.

7.7.8 Appendix

The material in the Appendix is not required for quizzes and assignments.

Motivation for Using NumPy: It is Fast!

Let's compare computing a vector dot product in Python (using lists) and compare it with NumPy's dot-product function. Mathematically, the dot product between two vectors $\mathbf{x}$ and $\mathbf{w}$ can be written as follows:

$$z = \sum_i x_i w_i = x_1 \times w_1 + x_2 \times w_2 + ... + x_n \times w_n = \mathbf{x}^\top \mathbf{w}$$

First, the Python implementation using a for-loop.

[97]:
```
def python_forloop_list_approach(x, w):
    z = 0.
    for i in range(len(x)):
        z += x[i] * w[i]
    return z


a = [1., 2., 3.]
b = [4., 5., 6.]

print(python_forloop_list_approach(a, b))
```

```
32.0
```

Let us compute the runtime for two larger (1000-element) vectors using IPython's `%timeit` magic function.

[98]:
```
large_a = list(range(1000))
large_b = list(range(1000))


%timeit python_forloop_list_approach(large_a, large_b)
```

```
120 µs ± 9.22 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Next, we use the `dot` function/method implemented in NumPy to compute the dot product between two vectors and run `%timeit` afterwards.

[99]:
```
def numpy_dotproduct_approach(x, w):
    # same as np.dot(x, w)
    # and same as x @ w
    return x.dot(w)


a = np.array([1., 2., 3.])
b = np.array([4., 5., 6.])

print(numpy_dotproduct_approach(a, b))
```

```
32.0
```

[100]:
```
large_a = np.arange(1000)
large_b = np.arange(1000)

%timeit numpy_dotproduct_approach(large_a, large_b)
```

```
1.58 µs ± 49 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

As we can see, replacing the for-loop with NumPy's dot function makes the computation of the vector dot product approximately 100 times faster.

7.7.9 References

1. Scientific Computing in Python: Introduction to NumPy and Matplotlib, by Sebastian Raschka, available at: https://sebastianraschka.com/blog/2020/numpy-intro.html.

BACK TO TOP

7.8 Lecture 8 - Data Manipulation with pandas

7.8.1 8.1 Introduction to pandas

pandas is a library designed for working with structured data, such as tabular data (e.g., from .csv files, Excel files) or relational databases (e.g., SQL).

> The **DataFrame** object in pandas is a 2-dimensional tabular, column-oriented data structure. The DataFrame is similar to an Excel spreadsheet and can store data of different types (including text characters, integers, floating-point values, categorical data, and more).

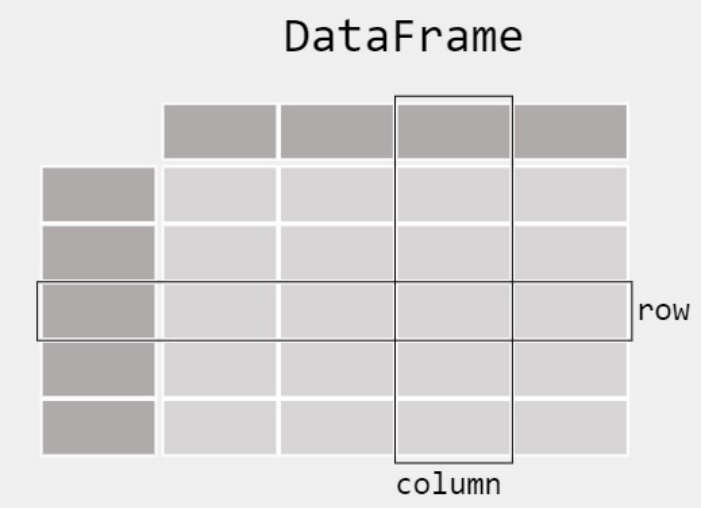


Figure source: Reference [2].

The pandas name is derived from the term *panel data*, which is a term from economics for multi-dimensional structured data.

7.8.2 8.2 Importing Data and Summary Statistics

Let's begin by importing the pandas package using the common abbreviation pd. Loading pandas as pd is standard practice.

[1]:
```
import pandas as pd
```

A wide range of input/output formats are supported by pandas:

- CSV, text
- SQL database
- Excel
- HDF5
- json
- html
- pickle
- sas, stata

- ...

For importing .csv files, the function `read_csv()` in `pandas` allows to easily import data. By default, it assumes that the data is comma-separated, but we can also specify the delimiter used in the data (e.g., tab, semicolon, etc.). There are several parameters that can be specified in `read_csv()`. See the documentation here.

Let's load the data in the file `country-total` located in the folder `data` and save it under the name `unemployment`. This file contains unemployment information for several countries over a time period.

The function `read_csv()` returns a `DataFrame`, as shown below. Note that the DataFrame has 20,796 rows, and the output of the cell displayed only the first 5 and last 5 rows (or the first and last 30, depending on your system), since displaying all 20,976 rows is probably not what we want at this point anyway.

[2]:
```
# Import data
unemployment = pd.read_csv('data/country_total.csv')
# Show the DataFrame
unemployment
```

[2]:
```
      country seasonality    month  unemployment  unemployment_rate
0          at         nsa  1993.01        171000                4.5
1          at         nsa  1993.02        175000                4.6
2          at         nsa  1993.03        166000                4.4
3          at         nsa  1993.04        157000                4.1
4          at         nsa  1993.05        147000                3.9
...       ...         ...      ...           ...                ...
20791      uk       trend  2010.06       2429000                7.7
20792      uk       trend  2010.07       2422000                7.7
20793      uk       trend  2010.08       2429000                7.7
20794      uk       trend  2010.09       2447000                7.8
20795      uk       trend  2010.10       2455000                7.8

[20796 rows x 5 columns]
```

We could have also used `print` to display the DataFrame.

[3]:
```
print(unemployment)
```

```
      country seasonality    month  unemployment  unemployment_rate
0          at         nsa  1993.01        171000                4.5
1          at         nsa  1993.02        175000                4.6
2          at         nsa  1993.03        166000                4.4
3          at         nsa  1993.04        157000                4.1
4          at         nsa  1993.05        147000                3.9
...       ...         ...      ...           ...                ...
20791      uk       trend  2010.06       2429000                7.7
20792      uk       trend  2010.07       2422000                7.7
20793      uk       trend  2010.08       2429000                7.7
20794      uk       trend  2010.09       2447000                7.8
20795      uk       trend  2010.10       2455000                7.8

[20796 rows x 5 columns]
```

When the DataFrames have a large number of rows that take large portion of the screen, we can inspect the data by using the `.head()` method. By default, this shows the **header** (names of the columns, commonly referred to as **column labels**) and the first five rows (having indices ranging from 0 to 4, in the first column in the table). The **indices** are also referred to as **row labels**.

[4]:
```
unemployment.head()
```

[4]:
```
  country seasonality    month  unemployment  unemployment_rate
0      at         nsa  1993.01        171000                4.5
1      at         nsa  1993.02        175000                4.6
2      at         nsa  1993.03        166000                4.4
3      at         nsa  1993.04        157000                4.1
4      at         nsa  1993.05        147000                3.9
```

Passing an integer number as an argument to `.head(n)` returns that number of rows. To see the last n rows, use `.tail(n)`.

[5]:
```
# show the last 8 rows
unemployment.tail(8)
```

[5]:
```
      country seasonality    month  unemployment  unemployment_rate
20788      uk       trend  2010.03       2437000                7.8
20789      uk       trend  2010.04       2419000                7.8
20790      uk       trend  2010.05       2419000                7.7
20791      uk       trend  2010.06       2429000                7.7
20792      uk       trend  2010.07       2422000                7.7
20793      uk       trend  2010.08       2429000                7.7
20794      uk       trend  2010.09       2447000                7.8
20795      uk       trend  2010.10       2455000                7.8
```

To find the number of rows in a DataFrame, you can use the `len()` function, as with Python lists and other sequences.

[6]:
```
len(unemployment)
```

[6]:
```
20796
```

Alternatively, we can use the `shape` attribute to find the numbers of rows and columns, as with NumPy arrays. The cell output is a tuple, showing that there are 20,796 rows and 5 columns. Note that the left-most column in the above table showing row indices is not part of the data.

[7]:
```
unemployment.shape
```

[7]:
```
(20796, 5)
```

A useful method that generates various summary statistics of a DataFrame is `.describe()`, as shown below.

Notice in the above cell that the DataFrame has 5 columns, but the first 2 columns (country and seasonality) have textual (strings) data, and therefore the summary statistics are shown only for the columns with numeric data (month, unemployment, and unemployment_rate). If `.describe()` is called on textual data only, it will return the count, number of unique values, and the most frequent value along with its count.

[8]:
```
unemployment.describe()
```

[8]:
```
              month  unemployment  unemployment_rate
count  20796.000000  2.079600e+04       19851.000000
mean    1999.401290  7.900818e+05           8.179764
std        7.483751  1.015280e+06           3.922533
min     1983.010000  2.000000e+03           1.100000
25%     1994.090000  1.400000e+05           5.200000
50%     2001.010000  3.100000e+05           7.600000
75%     2006.010000  1.262250e+06          10.000000
max     2010.120000  4.773000e+06          20.900000
```

It is also possible to calculate individual statistics, such as `.min()`, `.max ()`, or `.mean()`, instead of using summary statistics with `.describe()`.

[9]:
```
unemployment.min()
```

[9]:
```
country                  at
seasonality             nsa
month                1983.01
unemployment            2000
unemployment_rate        1.1
dtype: object
```

To view the data types for each column use the `dtypes` attribute of the *unemployment*. The data types in this case are strings (`object` type), floats (`float64` type), and integers (`int64` type).

[10]:
```
unemployment.dtypes
```

[10]:
```
country               object
seasonality           object
month                float64
unemployment           int64
unemployment_rate    float64
dtype: object
```

And one more way to get a summary of a DataFrame is by using `info()`.

[11]:
```
unemployment.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20796 entries, 0 to 20795
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   country            20796 non-null  object
 1   seasonality        20796 non-null  object
 2   month              20796 non-null  float64
 3   unemployment       20796 non-null  int64
 4   unemployment_rate  19851 non-null  float64
dtypes: float64(2), int64(1), object(2)
memory usage: 812.5+ KB
```

Import Data From a URL

Above, we imported the unemployment data using the function `read_csv` and a relative file path to the `data` directory. The function `read_csv` is very flexible and it also allows importing data using a URL as the file path.

A csv file with data on world countries and their abbreviations is located at https://raw.githubusercontent.com/dlab-berkeley/introduction-to-pandas/master/data/countries.csv

Using `read_csv`, we can import the country data and save it to the variable `countries`. This DataFrame has 30 rows.

[12]:
```
countries = pd.read_csv('https://raw.githubusercontent.com/dlab-berkeley/introduction-to-
↪pandas/master/data/countries.csv')
countries
```

[12]:
```
   country google_country_code country_group  \
0       at                   AT            eu
1       be                   BE            eu
2       bg                   BG            eu
3       hr                   HR        non-eu
4       cy                   CY            eu
5       cz                   CZ            eu
6       dk                   DK            eu
7       ee                   EE            eu
8       fi                   FI            eu
9       fr                   FR            eu
10      de                   DE            eu
11      gr                   GR            eu
12      hu                   HU            eu
13      ie                   IE            eu
14      it                   IT            eu
15      lv                   LV            eu
16      lt                   LT            eu
17      lu                   LU            eu
18      mt                   MT            eu
19      nl                   NL            eu
20      no                   NO        non-eu
21      pl                   PL            eu
22      pt                   PT            eu
23      ro                   RO            eu
24      sk                   SK            eu
25      si                   SI            eu
26      es                   ES            eu
27      se                   SE            eu
28      tr                   TR        non-eu
29      uk                   GB            eu

                                      name_en  \
0                                     Austria
1                                     Belgium
2                                    Bulgaria
3                                     Croatia
4                                      Cyprus
5                                Czech Republic
6                                     Denmark
7                                     Estonia
8                                     Finland
9                                      France
10  Germany (including  former GDR from 1991)
11                                     Greece
12                                    Hungary
13                                    Ireland
14                                      Italy
15                                     Latvia
16                                  Lithuania
17                                 Luxembourg
18                                      Malta
19                                Netherlands
```

(continues on next page)

(continued from previous page)

```
20                                    Norway
21                                    Poland
22                                  Portugal
23                                   Romania
24                                  Slovakia
25                                  Slovenia
26                                     Spain
27                                    Sweden
28                                    Turkey
29                            United Kingdom

                                          name_fr  \
0                                        Autriche
1                                        Belgique
2                                        Bulgarie
3                                         Croatie
4                                          Chypre
5                              République tchèque
6                                        Danemark
7                                         Estonie
8                                        Finlande
9                                          France
10  Allemagne (incluant l'ancienne RDA à partir de...
11                                         Grèce
12                                        Hongrie
13                                        Irlande
14                                         Italie
15                                       Lettonie
16                                       Lituanie
17                                    Luxembourg
18                                         Malte
19                                      Pays-Bas
20                                       Norvège
21                                       Pologne
22                                      Portugal
23                                      Roumanie
24                                     Slovaquie
25                                      Slovénie
26                                       Espagne
27                                         Suède
28                                       Turquie
29                                  Royaume-Uni

                                          name_de   latitude  longitude
0                                     Österreich  47.696554  13.345980
1                                        Belgien  50.501045   4.476674
2                                      Bulgarien  42.725674  25.482322
3                                       Kroatien  44.746643  15.340844
4                                         Zypern  35.129141  33.428682
5                          Tschechische Republik  49.803531  15.474998
6                                       Dänemark  55.939684   9.516689
7                                        Estland  58.592469  25.806950
```

(continues on next page)

(continued from previous page)

```
8                                              Finnland  64.950159  26.067564
9                                             Frankreich  46.710994   1.718561
10  Deutschland (einschließlich der ehemaligen DDR...  51.163825  10.454048
11                                         Griechenland  39.698467  21.577256
12                                              Ungarn  47.161163  19.504265
13                                               Irland  53.415260  -8.239122
14                                              Italien  42.504191  12.573787
15                                             Lettland  56.880117  24.606555
16                                              Litauen  55.173687  23.943168
17                                           Luxemburg  49.815319   6.133352
18                                               Malta  35.902422  14.447461
19                                          Niederlande  52.108118   5.330198
20                                             Norwegen  64.556460  12.665766
21                                               Polen  51.918907  19.134334
22                                             Portugal  39.558069  -7.844941
23                                             Rumänien  45.942611  24.990152
24                                             Slowakei  48.672644  19.700032
25                                            Slowenien  46.149259  14.986617
26                                              Spanien  39.895013  -2.988296
27                                             Schweden  62.198467  14.896307
28                                               Türkei  38.952942  35.439795
29                                 Vereinigtes Königreich  54.315447  -2.232612
```

Similar to the above example, we can use `shape` and `describe()` the understand the *countries* DataFrame. In this case `.describe()` is not very useful, because only 2 of the columns have numeric values.

[13]:
```
countries.shape
```

[13]:
```
(30, 8)
```

[14]:
```
# explore the countries data
countries.describe()
```

[14]:
```
        latitude  longitude
count  30.000000  30.000000
mean   49.092609  14.324579
std     7.956624  11.257010
min    35.129141  -8.239122
25%    43.230916   6.979186
50%    49.238087  14.941462
75%    54.090400  23.351690
max    64.950159  35.439795
```

Import Data from Excel File

In a similar way, we can import data from an Excel file using the function `read_excel()` .

[15]:
```
titanic = pd.read_excel('data/titanic.xls')
titanic
```

[15]:
```
    pclass  survived                                             name  \
0        1         1                    Allen, Miss. Elisabeth Walton
1        1         1                   Allison, Master. Hudson Trevor
2        1         0                     Allison, Miss. Helen Loraine
3        1         0             Allison, Mr. Hudson Joshua Creighton
4        1         0  Allison, Mrs. Hudson J C (Bessie Waldo Daniels)
5        1         1                                  Anderson, Mr. Harry
6        1         1                Andrews, Miss. Kornelia Theodosia
7        1         0                              Andrews, Mr. Thomas Jr
8        1         1    Appleton, Mrs. Edward Dale (Charlotte Lamson)
9        1         0                            Artagaveytia, Mr. Ramon
10       1         0                             Astor, Col. John Jacob
11       1         1  Astor, Mrs. John Jacob (Madeleine Talmadge Force)
12       1         1                     Aubart, Mme. Leontine Pauline
13       1         1                      Barber, Miss. Ellen "Nellie"
14       1         1             Barkworth, Mr. Algernon Henry Wilson
15       1         0                                Baumann, Mr. John D
16       1         0                           Baxter, Mr. Quigg Edmond
17       1         1  Baxter, Mrs. James (Helene DeLaudeniere Chaput)
18       1         1                              Bazzani, Miss. Albina
19       1         0                               Beattie, Mr. Thomson

       sex     age  sibsp  parch    ticket      fare    cabin embarked  boat  \
0   female      29      0      0     24160  211.3375       B5        S     2
1     male  0.9167      1      2    113781  151.5500  C22 C26        S    11
2   female       2      1      2    113781  151.5500  C22 C26        S  None
3     male      30      1      2    113781  151.5500  C22 C26        S  None
4   female      25      1      2    113781  151.5500  C22 C26        S  None
5     male      48      0      0     19952   26.5500      E12        S     3
6   female      63      1      0     13502   77.9583       D7        S    10
7     male      39      0      0    112050    0.0000      A36        S  None
8   female      53      2      0     11769   51.4792     C101        S     D
9     male      71      0      0  PC 17609   49.5042     None        C  None
10    male      47      1      0  PC 17757  227.5250  C62 C64        C  None
11  female      18      1      0  PC 17757  227.5250  C62 C64        C     4
12  female      24      0      0  PC 17477   69.3000      B35        C     9
13  female      26      0      0     19877   78.8500     None        S     6
14    male      80      0      0     27042   30.0000      A23        S     B
15    male    None      0      0  PC 17318   25.9250     None        S  None
16    male      24      0      1  PC 17558  247.5208  B58 B60        C  None
17  female      50      0      1  PC 17558  247.5208  B58 B60        C     6
18  female      32      0      0     11813   76.2917      D15        C     8
19    male      36      0      0     13050   75.2417       C6        C     A

    body                        home.dest
0   None                       St Louis, MO
1   None  Montreal, PQ / Chesterville, ON
```

(continues on next page)

(continued from previous page)

```
2   None  Montreal, PQ / Chesterville, ON
3    135  Montreal, PQ / Chesterville, ON
4   None  Montreal, PQ / Chesterville, ON
5   None                  New York, NY
6   None                    Hudson, NY
7   None                   Belfast, NI
8   None             Bayside, Queens, NY
9     22             Montevideo, Uruguay
10   124                  New York, NY
11  None                  New York, NY
12  None                 Paris, France
13  None                          None
14  None                 Hessle, Yorks
15  None                  New York, NY
16  None                  Montreal, PQ
17  None                  Montreal, PQ
18  None                          None
19  None                  Winnipeg, MN
```

Note that all DataFrames that we loaded so far in this Jupyter notebook, i.e., `unemployment`, `countries`, and `titanic`, are stored in the memory, and we can access them when needed once they are loaded. For example, we can check the shape of the `titanic` DataFrame.

[16]:
```
titanic.shape
```

[16]:
```
(20, 14)
```

Create a DataFrame

Alternatively, we can manually create DataFrames, instead of importing from a file. The following DataFrame called `simple_table` contains information from the `titanic` data. You can notice that the DataFrame is created similarly to creating a dictionary, where the column headers represent keys, and the data in each column are lists of values.

[17]:
```
simple_table = pd.DataFrame({
        "Name": ["Braund, Mr. Owen Harris",
                 "Allen, Mr. William Henry",
                 "Bonnell, Miss. Elizabeth"],
        "Age": [22, 35, 58],
        "Sex": ["male", "male", "female"]})
simple_table
```

[17]:
```
                        Name  Age     Sex
0   Braund, Mr. Owen Harris   22    male
1  Allen, Mr. William Henry   35    male
2  Bonnell, Miss. Elizabeth   58  female
```

[18]:
```
simple_table.shape
```

[18]:
```
(3, 3)
```

7.8.3 8.3 Rename, Index, and Slice

Let's look again at the *unemployment* DataFrame. You may have noticed that the `month` column actually includes the year and the month added as decimals (e.g., 1993.01 should be year 1993 and month 01).

[19]:
```
unemployment.head(3)
```

[19]:

	country	seasonality	month	unemployment	unemployment_rate
0	at	nsa	1993.01	171000	4.5
1	at	nsa	1993.02	175000	4.6
2	at	nsa	1993.03	166000	4.4

Let's rename the column into *year_month*. The `.rename()` method allows modifying column and/or row names. As you can see in the cell below, we passed a dictionary to the `columns` parameter, with the original name `month` as the key and the new name `year_month` as the value. Importantly, we also set the `inplace` parameter to `True`, which indicates that we want to modify the *actual* DataFrame, and not to create a new DataFrame.

[20]:
```
unemployment.rename(columns={'month' : 'year_month'}, inplace=True)
unemployment.head(3)
```

[20]:

	country	seasonality	year_month	unemployment	unemployment_rate
0	at	nsa	1993.01	171000	4.5
1	at	nsa	1993.02	175000	4.6
2	at	nsa	1993.03	166000	4.4

To observe the effect of `inplace=True`, let's run in the next cell another `.rename()` method to change the column `country` to `year`.

[21]:
```
unemployment.rename(columns={'country' : 'year'}).head(3)
```

[21]:

	year	seasonality	year_month	unemployment	unemployment_rate
0	at	nsa	1993.01	171000	4.5
1	at	nsa	1993.02	175000	4.6
2	at	nsa	1993.03	166000	4.4

The above code didn't change the actual `unemployment` DataFrame, as we can check that in the following cell. Instead, it created a copy of the `unemployment` DataFrame in which it changed the name of the column `country`.

[22]:
```
unemployment.head(3)
```

[22]:

	country	seasonality	year_month	unemployment	unemployment_rate
0	at	nsa	1993.01	171000	4.5
1	at	nsa	1993.02	175000	4.6
2	at	nsa	1993.03	166000	4.4

Selecting Columns

To select a single column of the DataFrame, we can either use the name of the column enclosed in square brackets `[]` or the dot notation `.` (i.e., via *attribute access*). It is preferable to use the square brackets notation, since a column name might inadvertently have the same name as an built-in `pandas` method.

[23]:
```
# access with square brackets
unemployment['year_month'].head()
```

[23]:
```
0    1993.01
1    1993.02
2    1993.03
3    1993.04
4    1993.05
Name: year_month, dtype: float64
```

[24]:
```
# access with dot notation
unemployment.year_month.head()
```

[24]:
```
0    1993.01
1    1993.02
2    1993.03
3    1993.04
4    1993.05
Name: year_month, dtype: float64
```

When selecting a single column, we obtain a `pandas` **Series** object, which is a single vector of data with an associated array of index row labels shown in the left-most column.

> A **Series** object in `pandas` represents an 1-dimensional vector of data (i.e., a column of data).

If we check the type of the `unemployment` object and `unemployment['year_month']` object, we can see that the first one is DataFrame and the second one is Series.

[25]:
```
type(unemployment)
```

[25]:
```
pandas.core.frame.DataFrame
```

[26]:
```
type(unemployment['year_month'])
```

[26]:
```
pandas.core.series.Series
```

`pandas` provide many methods that can be applied to `Series` objects. A few examples are shown below.

[27]:
```
print('minimum is ', unemployment['year_month'].min())
print('maximum is ', unemployment['year_month'].max())
print('mean value is ', unemployment['year_month'].mean())
```

```
minimum is  1983.01
maximum is  2010.12
mean value is  1999.4012896710906
```

To select multiple columns in `pandas`, use a list of column names within the selection brackets `[]`.

[28]:
```
unemployment[['country','year_month']].head()
```

[28]:

	country	year_month
0	at	1993.01
1	at	1993.02
2	at	1993.03
3	at	1993.04
4	at	1993.05

Selecting Rows

One way to select rows is by using the `[]` operator, similar to indexing and slicing in Python lists and other sequence data.

[29]:
```
unemployment[0:4]
```

[29]:
```
  country seasonality  year_month  unemployment  unemployment_rate
0      at         nsa     1993.01        171000                4.5
1      at         nsa     1993.02        175000                4.6
2      at         nsa     1993.03        166000                4.4
3      at         nsa     1993.04        157000                4.1
```

Another graphical representation of a DataFrame is shown in the figure below.

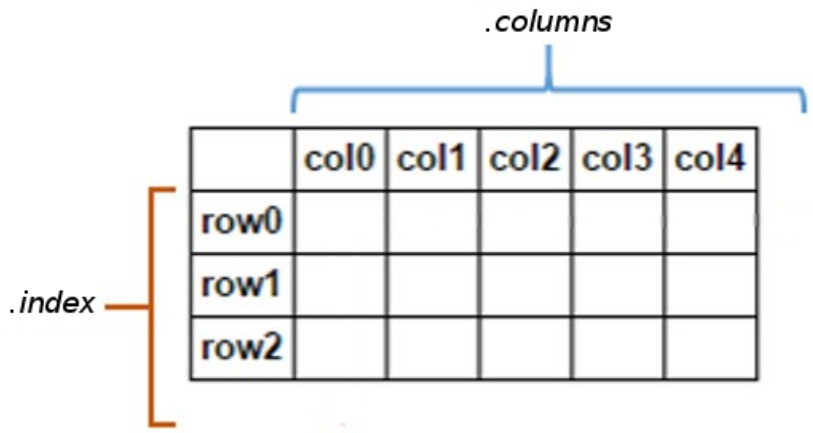


Figure source: Reference [2].

The first column with the indices in `pandas` DataFrames does not need to be a sequence of integers, but it can can also contains strings or other numeric data (e.g., dates, years).

For instance, let's create a `DataFrame` called *bacteria* to see how indexing with string indices works. We again pass in a dictionary, with the keys corresponding to column names and the values to the data, and in addition we pass a list of strings called `index`. (Compare to the *simple_table* above, which does not use `index` for creating the DataFrame, and in that case, the indices were automatically set to integer numbers.)

[30]:
```
bacteria = pd.DataFrame({'bacteria_counts' : [632, 1638, 569, 115],
                         'other_feature' : [438, 833, 234, 298]},
                         index=['Firmicutes', 'Proteobacteria', 'Actinobacteria',
↪'Bacteroidetes'])
bacteria
```

[30]:
```
               bacteria_counts  other_feature
Firmicutes                 632            438
Proteobacteria            1638            833
Actinobacteria             569            234
Bacteroidetes              115            298
```

For selecting rows and/or columns in `pandas`, beside the use of square brackets, two other operators are used: `.loc` and `.iloc`.

The operator `.loc` works with string locations (string labels), and `.iloc` works with integer locations (integer labels). These two operators can accept either a single label, a list of labels, or a slice of labels (e.g., `'a:f'` or `2:5`).

For instance, if we're interested in the row *Actinobacteria*, we can use `.loc` and the index name. This returns the column values for the specified row.

[31]:
```
bacteria.loc['Actinobacteria']
```

[31]:
```
bacteria_counts    569
other_feature      234
Name: Actinobacteria, dtype: int64
```

We could have also used "positional indexing" with square brackets `[]`, even though the indices are strings. The difference is that `.loc` returns a `Series` object because we selected a single label, while `[2:3]` returns a `DataFrame` because we selected a range of positions.

[32]:
```
bacteria[2:3]
```

[32]:
```
                bacteria_counts  other_feature
Actinobacteria              569            234
```

Let's return to the *unemployment* data to show how `.iloc` is used, since *unemployment* has integer indices. To select specific rows, we can do the following.

[33]:
```
unemployment.iloc[0:4]
```

[33]:
```
  country seasonality  year_month  unemployment  unemployment_rate
0      at         nsa     1993.01        171000                4.5
1      at         nsa     1993.02        175000                4.6
2      at         nsa     1993.03        166000                4.4
3      at         nsa     1993.04        157000                4.1
```

[34]:
```
unemployment.iloc[[1, 5, 6, 22]]
```

[34]:
```
   country seasonality  year_month  unemployment  unemployment_rate
1       at         nsa     1993.02        175000                4.6
5       at         nsa     1993.06        134000                3.5
6       at         nsa     1993.07        128000                3.4
22      at         nsa     1994.11        148000                3.9
```

We can also select a range of rows and specify the step value.

[35]:
```
unemployment.iloc[25:50:5]
```

[35]:
```
   country seasonality  year_month  unemployment  unemployment_rate
25      at         nsa     1995.02        174000                4.5
30      at         nsa     1995.07        123000                3.3
35      at         nsa     1995.12        175000                4.7
40      at         nsa     1996.05        159000                4.3
45      at         nsa     1996.10        146000                3.9
```

Selecting a Specific Value

Both `.loc` and `.iloc` can be used to select a particular value if they are given two arguments. The first argument is the row name (when using `.loc`) or the row index number (when using `.iloc`), while the second argument is the column name or index number.

Using `loc`, we can select "Bacteroidetes" and "bacteria_counts" to get the count of Bacteroidetes, as in the next cell below.

[36]:
```
bacteria
```

[36]:
```
                bacteria_counts  other_feature
Firmicutes                  632            438
Proteobacteria             1638            833
Actinobacteria              569            234
Bacteroidetes               115            298
```

[37]:
```
bacteria.loc['Bacteroidetes']['bacteria_counts']
```

[37]:
```
115
```

[38]:
```
# This is the same as above
bacteria.iloc[3][0]
```

[38]:
```
115
```

[39]:
```
# This the same as above
bacteria.iloc[3]['bacteria_counts']
```

[39]:
```
115
```

Or, for the *unemployment* data:

[40]:
```
# The year_month in the first row
unemployment.iloc[0,2]
```

[40]:
```
1993.01
```

[41]:
```
# This the same as above
unemployment.iloc[0][2]
```

[41]:
```
1993.01
```

Selecting Multiple Rows and Columns

Both `.loc` and `.iloc` can be used to select subsets of rows and columns at the same time if they are given lists as their two arguments, or slices for `.iloc`.

[42]:
```
unemployment.iloc[2:6,0:2]
```

[42]:
```
  country seasonality
2      at         nsa
3      at         nsa
4      at         nsa
5      at         nsa
```

Using `.iloc` on the *unemployment* DataFrame, select the rows starting at row 2 and ending at row 5, and the 0th, 2nd, and 3rd columns.

[43]:
```
unemployment.iloc[2:6,[0,2,3]]
```

[43]:
```
  country  year_month  unemployment
2      at     1993.03        166000
3      at     1993.04        157000
4      at     1993.05        147000
5      at     1993.06        134000
```

The same selection can be achieved by using the `.loc` operator and listing the column names.

[44]:
```
# The same as above
unemployment.loc[2:6,['country', 'year_month', 'unemployment']]
```

[44]:
```
  country  year_month  unemployment
2      at     1993.03        166000
3      at     1993.04        157000
4      at     1993.05        147000
5      at     1993.06        134000
6      at     1993.07        128000
```

We can also display values from a DataFrame that satisfy certain criteria using conditional expressions, such as <, >, ==, !=, etc.

One example is shown below where only the rows that have an unemployment rate greater than 15 are shown.

[45]:
```
unemployment[unemployment['unemployment_rate'] > 15].head(10)
```

[45]:
```
     country seasonality  year_month  unemployment  unemployment_rate
1717      bg         nsa     2000.02        523000               15.4
1718      bg         nsa     2000.03        547000               16.0
1719      bg         nsa     2000.04        560000               16.3
1720      bg         nsa     2000.05        561000               16.3
1721      bg         nsa     2000.06        554000               16.2
1722      bg         nsa     2000.07        558000               16.3
1723      bg         nsa     2000.08        569000               16.7
1724      bg         nsa     2000.09        574000               16.8
1725      bg         nsa     2000.10        583000               17.1
1726      bg         nsa     2000.11        597000               17.5
```

Differences between `loc` and `iloc`

To show the differences between `loc` and `iloc` let's consider the following example.

[46]:
```
df1 = pd.DataFrame({'x':[10, 20, 30, 40 ,50],
                    'y':[20, 30, 40, 50, 60],
                    'z':[30, 40, 50, 60, 70]},
                      index=[10, 11,12, 0, 1])
df1
```

[46]:
```
     x   y   z
10  10  20  30
11  20  30  40
```

(continues on next page)

(continued from previous page)

```
12  30  40  50
0   40  50  60
1   50  60  70
```

Note in the following cells that `iloc` selects the row with index location `0`, whereas `loc` selects the row with index label `0`.

[47]:
```
# value at index location 0
df1.iloc[0]
```

[47]:
```
x    10
y    20
z    30
Name: 10, dtype: int64
```

[48]:
```
# value at index label 0
df1.loc[0]
```

[48]:
```
x    40
y    50
z    60
Name: 0, dtype: int64
```

Also, there is a difference in the selected rows when using slicing operations with `iloc` and `loc`. One must be careful when using these operators, and always check the output to ensure it is as expected.

[49]:
```
# rows at index location between 0 and 1 (exclusive)
df1.iloc[0:1]
```

[49]:
```
     x   y   z
10  10  20  30
```

[50]:
```
# rows at index labels between 0 and 1 (inclusive)
df1.loc[0:1]
```

[50]:
```
    x   y   z
0  40  50  60
1  50  60  70
```

7.8.4 8.4 Creating New Columns, Reordering

Since the `year_month` column is not shown correctly, let's try to split it into two separate columns for years and months.

In the previous section we saw that the data type in this column is `float64`. We'll first extract the year using the `.astype()` method. This allows for type casting, i.e., using `.astype(int)` we will convert the floating point values into integer numbers (by truncating the decimals).

The new column *year* will be added on the right of the DataFrame.

[51]:
```
unemployment['year'] = unemployment['year_month'].astype(int)
unemployment.head()
```

[51]:
```
  country seasonality  year_month  unemployment  unemployment_rate  year
0      at         nsa     1993.01        171000                4.5  1993
```

(continues on next page)

(continued from previous page)

```
1       at          nsa      1993.02         175000                 4.6  1993
2       at          nsa      1993.03         166000                 4.4  1993
3       at          nsa      1993.04         157000                 4.1  1993
4       at          nsa      1993.05         147000                 3.9  1993
```

Next, let's create a new column *month*. We will subtract the *year* value from *year_month* to get the decimal portion of the value, and multiply the result by 100 and convert to `int`. Because of the truncating that occurs when casting to `int`, we first need to round the values to the nearest whole number using `round()`.

[52]:
```
unemployment['month'] = ((unemployment['year_month'] - unemployment['year']) * 100).
↪round(0).astype(int)
unemployment.head(12)
```

[52]:
```
   country seasonality  year_month  unemployment  unemployment_rate  year  \
0       at          nsa      1993.01         171000                4.5  1993
1       at          nsa      1993.02         175000                4.6  1993
2       at          nsa      1993.03         166000                4.4  1993
3       at          nsa      1993.04         157000                4.1  1993
4       at          nsa      1993.05         147000                3.9  1993
5       at          nsa      1993.06         134000                3.5  1993
6       at          nsa      1993.07         128000                3.4  1993
7       at          nsa      1993.08         130000                3.4  1993
8       at          nsa      1993.09         132000                3.5  1993
9       at          nsa      1993.10         141000                3.7  1993
10      at          nsa      1993.11         156000                4.1  1993
11      at          nsa      1993.12         169000                4.4  1993

    month
0       1
1       2
2       3
3       4
4       5
5       6
6       7
7       8
8       9
9      10
10     11
11     12
```

Now, let's try to reorder the newly created *year* and *month* columns in the DataFrame. For this, we will use the square brackets notation again, passing in a list of column names in the order we would like to see them.

[53]:
```
unemployment = unemployment[['country', 'seasonality',
                             'year_month', 'year', 'month',
                             'unemployment', 'unemployment_rate']]
unemployment.head(10)
```

[53]:
```
  country seasonality  year_month  year  month  unemployment  \
0      at          nsa      1993.01  1993      1        171000
1      at          nsa      1993.02  1993      2        175000
2      at          nsa      1993.03  1993      3        166000
```

(continues on next page)

(continued from previous page)

```
3      at         nsa     1993.04  1993      4         157000
4      at         nsa     1993.05  1993      5         147000
5      at         nsa     1993.06  1993      6         134000
6      at         nsa     1993.07  1993      7         128000
7      at         nsa     1993.08  1993      8         130000
8      at         nsa     1993.09  1993      9         132000
9      at         nsa     1993.10  1993     10         141000

   unemployment_rate
0                4.5
1                4.6
2                4.4
3                4.1
4                3.9
5                3.5
6                3.4
7                3.4
8                3.5
9                3.7
```

7.8.5 8.5 Merging

If we examine the `unemployment` DataFrame we can notice that we don't exactly know what the values in the `country` column refer to. We can fix that by getting the country names from the `countries` DataFrame that we imported earlier.

We can see in the `countries` data that *at* stands for Austria. This DataFrame even provides the country names in three different languages.

[54]:
```
countries.head()
```

[54]:
```
  country google_country_code country_group   name_en   name_fr     name_de  \
0      at                   AT            eu   Austria  Autriche  Österreich
1      be                   BE            eu   Belgium  Belgique     Belgien
2      bg                   BG            eu  Bulgaria  Bulgarie   Bulgarien
3      hr                   HR        non-eu   Croatia   Croatie    Kroatien
4      cy                   CY            eu    Cyprus    Chypre      Zypern

    latitude  longitude
0  47.696554  13.345980
1  50.501045   4.476674
2  42.725674  25.482322
3  44.746643  15.340844
4  35.129141  33.428682
```

Because the data we need is stored in two separate files, we will first merge the two DataFrames. The `country` column is shown in both DataFrames, so it is a good option for joining the data. However, we don't need all columns in the `countries` DataFrame, and therefore, we will create a new DataFrame. To select certain columns to retain, we can use the bracket notation that we used earlier to reorder the columns.

[55]:
```
country_names = countries[['country', 'country_group', 'name_en']]
```

[56]:
```
country_names.head(5)
```

[56]:
```
  country country_group   name_en
0      at            eu   Austria
1      be            eu   Belgium
2      bg            eu  Bulgaria
3      hr        non-eu   Croatia
4      cy            eu    Cyprus
```

`pandas` include an easy-to-use `merge` function, which has the following syntax.

```
pd.merge(first_file, second_file, on=['column_name'])
```

[57]:
```
unemployment = pd.merge(unemployment, country_names, on=['country'])
unemployment.head()
```

[57]:
```
  country seasonality  year_month  year  month  unemployment  \
0      at         nsa     1993.01  1993      1        171000
1      at         nsa     1993.02  1993      2        175000
2      at         nsa     1993.03  1993      3        166000
3      at         nsa     1993.04  1993      4        157000
4      at         nsa     1993.05  1993      5        147000

   unemployment_rate country_group  name_en
0                4.5            eu  Austria
1                4.6            eu  Austria
2                4.4            eu  Austria
3                4.1            eu  Austria
4                3.9            eu  Austria
```

If we want to merge two files using multiple columns that exist in both files, we can pass a list of column names to the on parameter.

For more information on merging, check the pandas documentation.

7.8.6 8.6 Calculating Unique and Missing Values

In the *unemployment* DataFrame, we might want to know what countries we have data for. To extract this information, we can use the `.unique()` method. Note that the countries are listed in the right-most column `name-en` so we will use it to find the unique elements in that column.

[58]:
```
unemployment.head()
```

[58]:
```
  country seasonality  year_month  year  month  unemployment  \
0      at         nsa     1993.01  1993      1        171000
1      at         nsa     1993.02  1993      2        175000
2      at         nsa     1993.03  1993      3        166000
3      at         nsa     1993.04  1993      4        157000
4      at         nsa     1993.05  1993      5        147000

   unemployment_rate country_group  name_en
0                4.5            eu  Austria
1                4.6            eu  Austria
2                4.4            eu  Austria
```

(continues on next page)

(continued from previous page)

```
3                4.1            eu  Austria
4                3.9            eu  Austria
```

[59]:
```
unemployment['name_en'].unique()
## We can also
# unemployment.name_en.unique()
```

[59]:
```
array(['Austria', 'Belgium', 'Bulgaria', 'Cyprus', 'Czech Republic',
       'Germany (including  former GDR from 1991)', 'Denmark', 'Estonia',
       'Spain', 'Finland', 'France', 'Greece', 'Croatia', 'Hungary',
       'Ireland', 'Italy', 'Lithuania', 'Luxembourg', 'Latvia', 'Malta',
       'Netherlands', 'Norway', 'Poland', 'Portugal', 'Romania', 'Sweden',
       'Slovenia', 'Slovakia', 'Turkey', 'United Kingdom'], dtype=object)
```

To get a count of the **number of unique countries,** we can use the `.nunique()` method.

[60]:
```
unemployment['name_en'].nunique()
```

[60]:
```
30
```

Or, we can also use `len()` to get the number of items in the above array.

[61]:
```
len(unemployment['name_en'].unique())
```

[61]:
```
30
```

If we are interested to know **how many observations** there are per country, `pandas` has a method called `.value_counts()` that returns the counts for the unique values in a column.

[62]:
```
unemployment['name_en'].value_counts()
```

[62]:
```
France                                      1008
Sweden                                      1008
Portugal                                    1008
Netherlands                                 1008
Luxembourg                                  1008
Denmark                                     1008
Belgium                                     1008
Spain                                       1008
Ireland                                     1008
United Kingdom                              1002
Italy                                        924
Finland                                      828
Norway                                       786
Austria                                      648
Hungary                                      576
Slovakia                                     576
Slovenia                                     576
Bulgaria                                     576
Malta                                        576
Poland                                       576
Germany (including  former GDR from 1991)    504
Czech Republic                               468
Latvia                                       459
```

(continues on next page)

(continued from previous page)

```
Lithuania                                459
Greece                                   450
Romania                                  423
Cyprus                                   396
Estonia                                  387
Croatia                                  324
Turkey                                   210
Name: name_en, dtype: int64
```

By default, the output is sorted by values. If we would like it sorted by index (or, by country name in this case since the countries were listed in alphabetical order), we can append the `.sort_index()` method.

[63]:
```
unemployment['name_en'].value_counts().sort_index()
```

[63]:
```
Austria                                  648
Belgium                                 1008
Bulgaria                                 576
Croatia                                  324
Cyprus                                   396
Czech Republic                           468
Denmark                                 1008
Estonia                                  387
Finland                                  828
France                                  1008
Germany (including  former GDR from 1991)    504
Greece                                   450
Hungary                                  576
Ireland                                 1008
Italy                                    924
Latvia                                   459
Lithuania                                459
Luxembourg                              1008
Malta                                    576
Netherlands                             1008
Norway                                   786
Poland                                   576
Portugal                                1008
Romania                                  423
Slovakia                                 576
Slovenia                                 576
Spain                                   1008
Sweden                                  1008
Turkey                                   210
United Kingdom                          1002
Name: name_en, dtype: int64
```

As we noticed earlier, there are missing values in the `unemployment_rate` column. To find out **how many unemployment rate values are missing** we will use the `.isnull()` method, which returns a corresponding boolean value for each missing entry in the `unemployment_rate` column. As we know, in Python `True` is equivalent to 1 and `False` is equivalent to 0. Thus, when we add the result with `.sum()`, we get a count for the total number of missing values.

[64]:
```
unemployment['unemployment_rate'].isnull().sum()
```

[64]:
```
945
```

GroupBy

If we would like to know how many missing values exist at the *country* level, we can take the main part of what we had above and create a new column in the `DataFrame`. This is the last column, in which `False` means that the value is not missing.

[65]:
```
unemployment['unemployment_rate_null'] = unemployment['unemployment_rate'].isnull()
unemployment.head()
```

[65]:
```
  country seasonality  year_month  year  month  unemployment  \
0      at         nsa     1993.01  1993      1        171000
1      at         nsa     1993.02  1993      2        175000
2      at         nsa     1993.03  1993      3        166000
3      at         nsa     1993.04  1993      4        157000
4      at         nsa     1993.05  1993      5        147000

   unemployment_rate country_group  name_en  unemployment_rate_null
0                4.5            eu  Austria                   False
1                4.6            eu  Austria                   False
2                4.4            eu  Austria                   False
3                4.1            eu  Austria                   False
4                3.9            eu  Austria                   False
```

To count the **number of missing values for each country,** we can use the `.groupby()` method. It groups the data by the country `name_en` column included in the parentheses, and the `.sum()` operation is performed over the `unemployment_rate_null` column.

[66]:
```
unemployment.groupby('name_en')['unemployment_rate_null'].sum()
```

[66]:
```
name_en
Austria                                      0
Belgium                                      0
Bulgaria                                   180
Croatia                                    216
Cyprus                                       0
Czech Republic                               0
Denmark                                      0
Estonia                                      0
Finland                                      0
France                                       0
Germany (including  former GDR from 1991)    0
Greece                                       0
Hungary                                     36
Ireland                                      0
Italy                                        0
Latvia                                       0
Lithuania                                    0
Luxembourg                                   0
Malta                                      180
Netherlands                                  0
Norway                                       0
```

(continues on next page)

(continued from previous page)

```
Poland                                    72
Portugal                                   0
Romania                                    0
Slovakia                                 108
Slovenia                                  36
Spain                                    117
Sweden                                     0
Turkey                                     0
United Kingdom                             0
Name: unemployment_rate_null, dtype: int64
```

7.8.7 8.7 Exporting A DataFrame to csv

If we wanted to save the last DataFrame as a .csv file, we can use the `.to_csv()` method.

[67]:
```
unemployment.to_csv('data/unemployment_missing.csv')
```

The file will be saved in the `data` directory.

By default, this method writes the indices in the column 0 (i.e., row labels). We probably don't want a column 0 with indices to be added, and we can set `index` to `False`. We can also specify the type of delimiter that we want to use, such as commas `(,)`, pipes (`|`), semicolons (`;`), tabs (`\t`), etc.

[68]:
```
unemployment.to_csv('data/unemployment_missing.csv', index=False, sep=',')
```

Now that we have the missing values saved, we can *drop* the last column `unemployment_rate_null` that we added to `unemployment`, as shown in the next cell below.

[69]:
```
unemployment.head()
```

[69]:
```
  country seasonality  year_month  year  month  unemployment  \
0      at         nsa     1993.01  1993      1        171000
1      at         nsa     1993.02  1993      2        175000
2      at         nsa     1993.03  1993      3        166000
3      at         nsa     1993.04  1993      4        157000
4      at         nsa     1993.05  1993      5        147000

   unemployment_rate country_group  name_en  unemployment_rate_null
0                4.5            eu  Austria                   False
1                4.6            eu  Austria                   False
2                4.4            eu  Austria                   False
3                4.1            eu  Austria                   False
4                3.9            eu  Austria                   False
```

[70]:
```
unemployment.drop('unemployment_rate_null', axis=1, inplace=True)
unemployment.head()
```

[70]:
```
  country seasonality  year_month  year  month  unemployment  \
0      at         nsa     1993.01  1993      1        171000
1      at         nsa     1993.02  1993      2        175000
2      at         nsa     1993.03  1993      3        166000
3      at         nsa     1993.04  1993      4        157000
```

(continues on next page)

(continued from previous page)

```
4      at         nsa     1993.05  1993      5        147000

   unemployment_rate country_group  name_en
0                4.5            eu  Austria
1                4.6            eu  Austria
2                4.4            eu  Austria
3                4.1            eu  Austria
4                3.9            eu  Austria
```

It is important to specify the `axis` parameter in the above code, where `axis=1` refers to columns (`axis=0` refers to rows). Note again that the parameter `inplace=True` modifies the actual DataFrame rather than returning a new DataFrame.

7.8.8 8.8 Dealing With Missing Values: Boolean Indexing

There are two main options for dealing with missing values:

- Fill the missing values with some other values.
- Remove the observations with missing values.

Here we will adopt the second approach and **exclude missing values** from our primary analyses. Instead of just getting rid of that data, it might make sense to store it in a separate DataFrame. This way, we could answer questions such as, "do missing values occur during certain months (or years) more frequently?"

To do this, we will use *boolean indexing* for filtering data.

Recall that `unemployment['unemployment_rate'].isnull()` produces an array of Boolean values. We used this previously when counting the number of missing values, and it is shown in the next cell.

[71]:
```
unemployment['unemployment_rate'].isnull()[:10]
```

[71]:
```
0    False
1    False
2    False
3    False
4    False
5    False
6    False
7    False
8    False
9    False
Name: unemployment_rate, dtype: bool
```

To create `unemployment_rate_missing`, we will index `unemployment` with the Boolean array above. This returns only the rows where the value in the array is `True`.

[72]:
```
unemployment_rate_missing = unemployment[unemployment['unemployment_rate'].isnull()]
unemployment_rate_missing.head()
```

[72]:
```
     country seasonality  year_month  year  month  unemployment  \
1656      bg         nsa     1995.01  1995      1        391000
1657      bg         nsa     1995.02  1995      2        387000
1658      bg         nsa     1995.03  1995      3        378000
1659      bg         nsa     1995.04  1995      4        365000
```

(continues on next page)

(continued from previous page)

```
1660      bg         nsa     1995.05  1995      5        346000

      unemployment_rate country_group   name_en
1656                NaN            eu  Bulgaria
1657                NaN            eu  Bulgaria
1658                NaN            eu  Bulgaria
1659                NaN            eu  Bulgaria
1660                NaN            eu  Bulgaria
```

It is also possible to specify multiple conditions using the & operator, but each condition needs to be inside of parentheses.

Now, to remove the missing data in `unemployment`, we can use the `.dropna()` method. This method drops all observations for which `unemployment_rate == NaN`.

[73]:
```
unemployment.dropna(subset=['unemployment_rate'], inplace=True)
unemployment.head()
```

[73]:
```
  country seasonality  year_month  year  month  unemployment  \
0      at         nsa     1993.01  1993      1        171000
1      at         nsa     1993.02  1993      2        175000
2      at         nsa     1993.03  1993      3        166000
3      at         nsa     1993.04  1993      4        157000
4      at         nsa     1993.05  1993      5        147000

   unemployment_rate country_group  name_en
0                4.5            eu  Austria
1                4.6            eu  Austria
2                4.4            eu  Austria
3                4.1            eu  Austria
4                3.9            eu  Austria
```

[74]:
```
# Check the shape of the modified DataFrame
unemployment.shape
```

[74]:
```
(19851, 9)
```

Sorting Values

If we want to know what the highest unemployment rates were, we can use the `.sort_values()` method to *sort the data.*

This cell sorted the data in *descending* order, and printed the first 10 rows.

[75]:
```
unemployment.sort_values('unemployment_rate', ascending=False)[:10]
```

[75]:
```
      country seasonality  year_month  year  month  unemployment  \
15526      pl         nsa     2004.02  2004      2       3531000
15525      pl         nsa     2004.01  2004      1       3520000
15514      pl         nsa     2003.02  2003      2       3460000
5663       es          sa     2010.09  2010      9       4773000
15527      pl         nsa     2004.03  2004      3       3475000
5999       es       trend     2010.09  2010      9       4769000
```

(continues on next page)

(continued from previous page)

```
6000       es       trend     2010.10  2010     10       4760000
15513      pl         nsa     2003.01  2003      1       3466000
5664       es          sa     2010.10  2010     10       4758000
5998       es       trend     2010.08  2010      8       4742000

       unemployment_rate country_group name_en
15526               20.9            eu  Poland
15525               20.7            eu  Poland
15514               20.7            eu  Poland
5663                20.6            eu   Spain
15527               20.6            eu  Poland
5999                20.6            eu   Spain
6000                20.6            eu   Spain
15513               20.6            eu  Poland
5664                20.6            eu   Spain
5998                20.5            eu   Spain
```

Several additional functionalities of `pandas` will be described in the next lectures.

7.8.9 References

1. Introduction to Pandas, Python Data Wrangling by D-Lab at UC Berkley, available at: https://github.com/dlab-berkeley/introduction-to-pandas.
2. Pandas documentation, available at https://pandas.pydata.org/pandas-docs/stable/.

BACK TO TOP

7.9 Lecture 9 - Data Visualization with Matplotlib

7.9.1 9.1 Introduction to Matplotlib

Matplotlib is a plotting library for Python created by John D. Hunter in 2003. Matplotlib is among the most widely used plotting libraries and is often the preferred choice for many data scientists and machine learning practitioners. The Matplotlib gallery on the official website at https://matplotlib.org/gallery/index.html contains code examples for creating various kinds of plots.

Matplotlib has two general interfaces for plotting: a **state-based approach** that is similar to Matlab's way of plotting, and a more Pythonic **object-oriented approach**. We will start with discussing the state-based approach, and continue afterward with the object-oriented approach.

The main plotting functions of Matplotlib are contained in the `pyplot` module, which is almost always imported as `plt`.

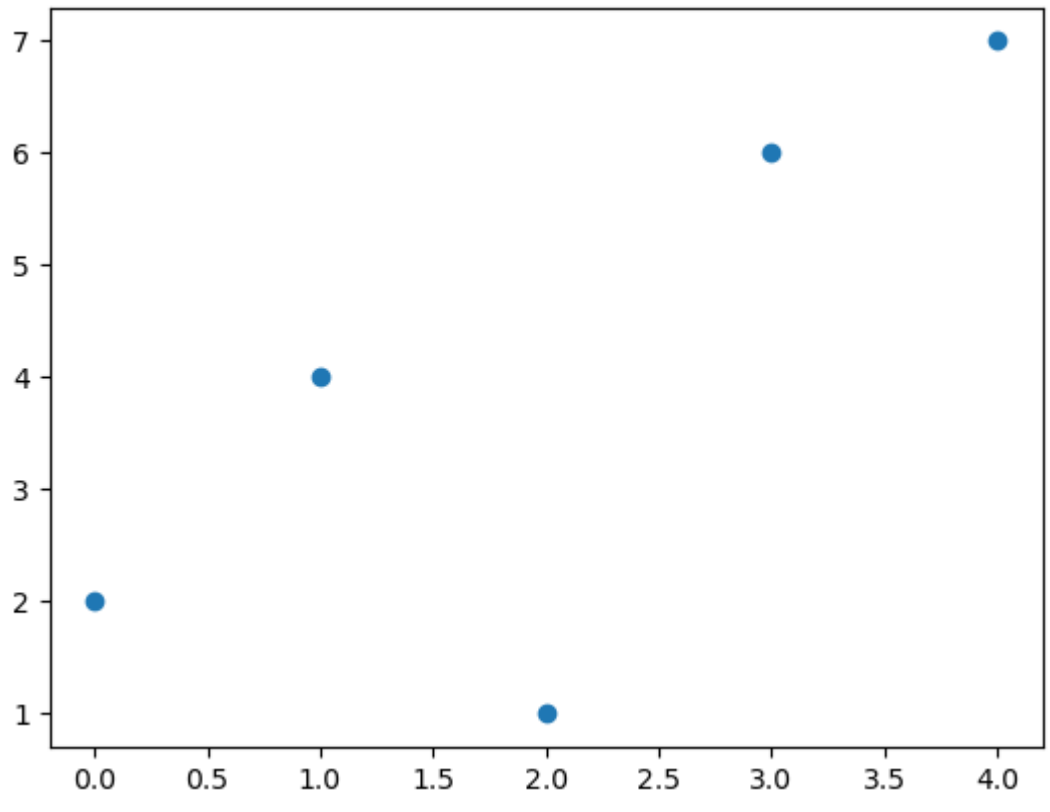
[1]:
```
import matplotlib.pyplot as plt
```

7.9.2 9.2 The State-based Approach

As we mentioned, the state-based approach was developed based on the way plotting is done in Matlab. I.e., we call different functions that each take care of an aspect of the plot.

Let's create a few plots to show how the state-based approach works.

[2]:
```
y = [2, 4, 1, 6, 7]

plt.plot(y, 'o')  # plot the data
plt.show()  # this line visualizes the plot
```



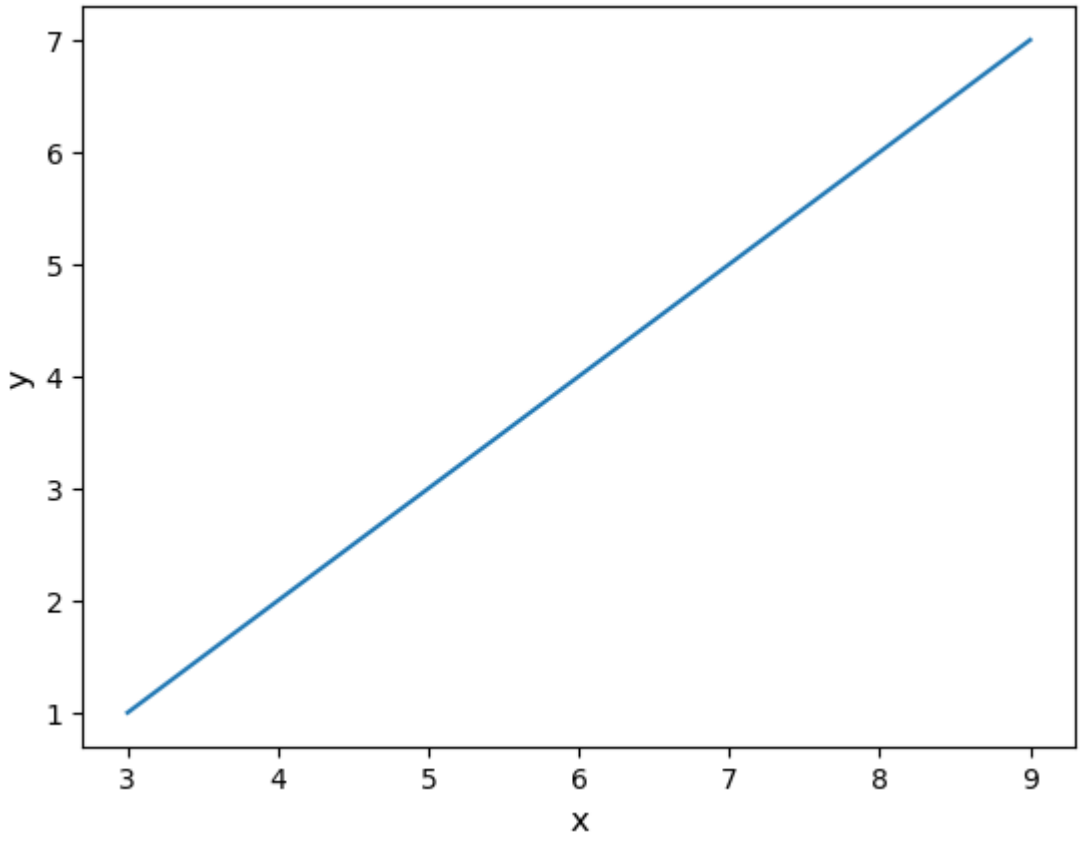
As you can see, the state-based approach includes a function call to `plt.plot` and afterward we call `plt.show` to show the plot in the notebook (or if we run this code from a script, the plot will show in an external image viewer). Note that technically the `plt.show` call is not necessary to render the plot in Jupyter notebooks, but it is recommended to do it anyway as it is good practice.

In the above plot we provided only the list `y`, and Matplotlib plotted each value using the indices in the list for the x-axis. I.e., the value 4 has coordinate 1 on the x-axis, and value 6 has coordinate 3 on the x-axis. The string `'o'` in `plt.plot` indicates that we didn't want to plot a line, but we wanted to plot the values with circle markers.

Also, when working with Jupyter notebooks, each line of code that is part of the same plot should be defined in the same cell. Otherwise, the elements of the plot won't be included in the same figure.

If we pass two arguments to `plt.plot`, then the first argument will represent the positions on the x-axis, and the second argument will represent the positions on the y-axis. This is shown in the next cell.

[3]:
```
x = [3, 4, 5, 6, 7, 8, 9]
y = [1, 2, 3, 4, 5, 6, 7]

plt.plot(x, y)  # plot the data
plt.xlabel('x', fontsize=12)  # set the x-axis label
plt.ylabel('y', fontsize=12)  # set the y-axis label
plt.show()  # this line visualizes the plot
```



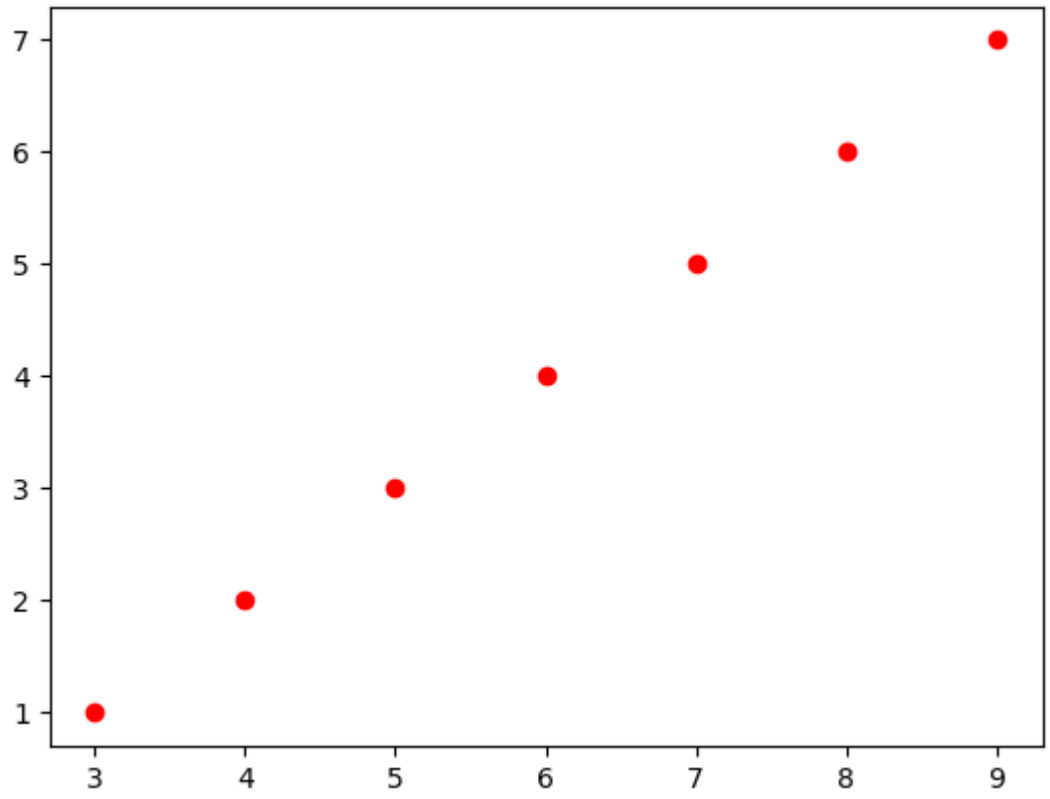
Axis Labels

When graphing data, it is always a good idea to provide **axis labels**. In the above graph, we used `plt.xlabel` and `plt.ylabel` to provide the corresponding labels for the x-axis and y-axis. And, we can also control the size of axes labels by assigning a value to the keyword `fontsize`.

Controlling Line Properties

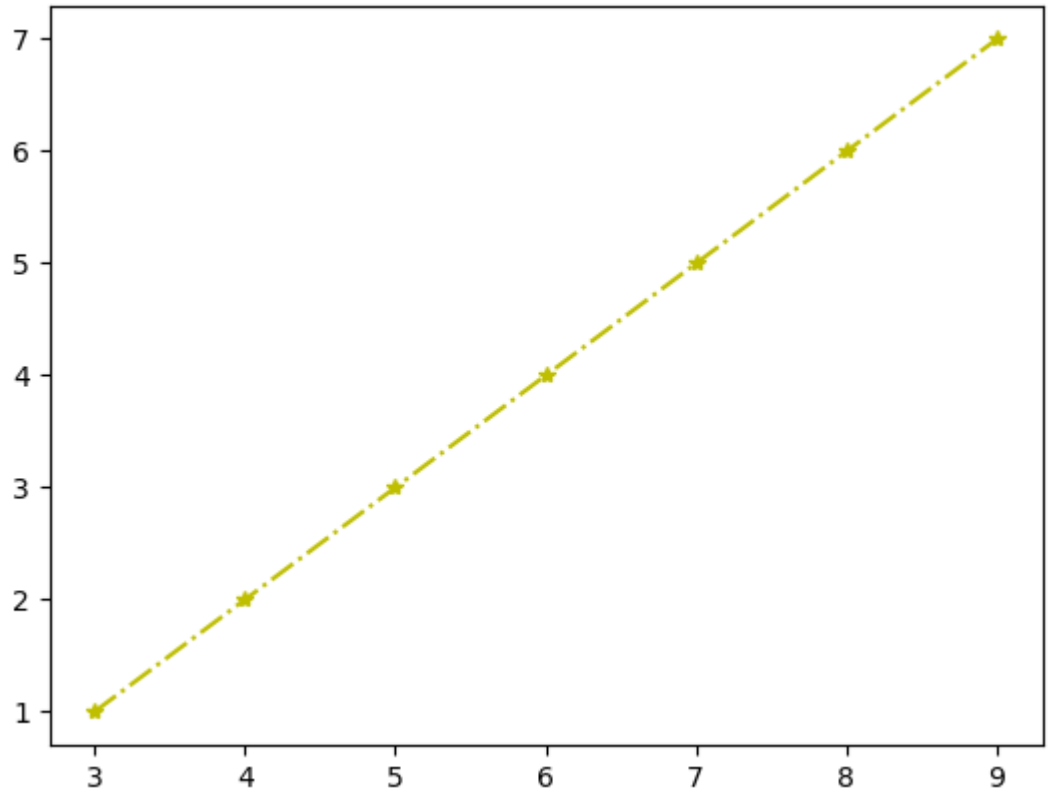
The `plt.plot` function is the most basic function to create any plot of paired data points `(x, y)`. By default, it creates a line plot (as shown above), but there are many optional parameters in `plt.plot` allowing to create many different variations. For example, instead of a line, we can plot the data as separate red markers by specifying the marker **format** in the third argument `'o'` to indicate points/circles as markers, and the **color** by setting the argument `c` for color to `'red'`.

[4]:
```
plt.plot(x, y, 'o', c='red')
plt.show()

## This above plt.plot function is equivalent to:
# plt.plot(x, y, marker='o', color='red', linestyle='')
## And is also equivalent to:
# plt.plot(x, y, 'ro')
```



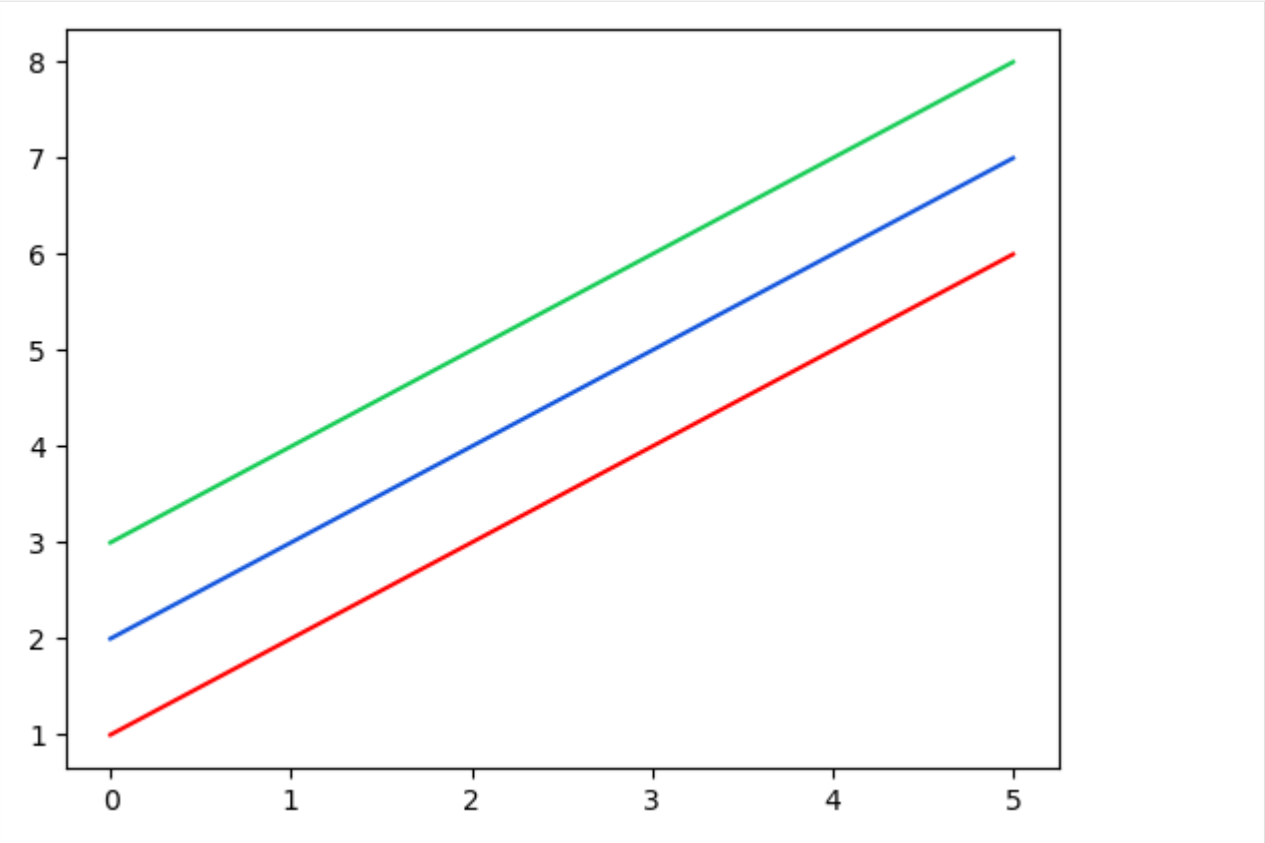
In the next cell, the third argument is related to the line format and may be used to specify three things at once: whether we want *markers* (and which type of marker), whether we want a *line* (and which type of line), and which *color* the markers/line should have. So, to create a blue line, you may specify `'-b'`. To create a yellow (`'y'`) dashes-dots line (`'-.'`) with star markers (`'*'`), we can use `'*-.y'`.

[5]:
```
plt.plot(x, y, '*-.y')
plt.show()
```



Matplotlib also allows defining **colors** by using the `color` keyword argument, where several basic colors can be defined with their names (e.g., "red" below), or many colors can be definted with their RGB hex codes.

[6]:
```
import numpy as np
x = np.linspace(0, 5, 10)

plt.plot(x, x+1, color="red")
plt.plot(x, x+2, color="#1155dd")        # RGB hex code for a bluish color
plt.plot(x, x+3, color="#15cc55")        # RGB hex code for a greenish color
plt.show()
```



The supported color abbreviations with the single letter codes include:

character	color
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

The formats for line styles include the following:

character	description
-	solid line style
–	dashed line style
-.	dash-dot line style
:	dotted line style

The following symbols can be used to specify markers:

character	description
.	point marker
,	pixel marker
o	circle marker
v	triangle_down marker
^	triangle_up marker
<	triangle_left marker
>	triangle_right marker
1	tri_down marker
2	tri_up marker
3	tri_left marker
4	tri_right marker
s	square marker
p	pentagon marker
*	star marker
h	hexagon1 marker
H	hexagon2 marker
+	plus marker
x	x marker
D	diamond marker
d	thin_diamond marker
‘	vline marker
_	hline marker

Line and Marker Styles

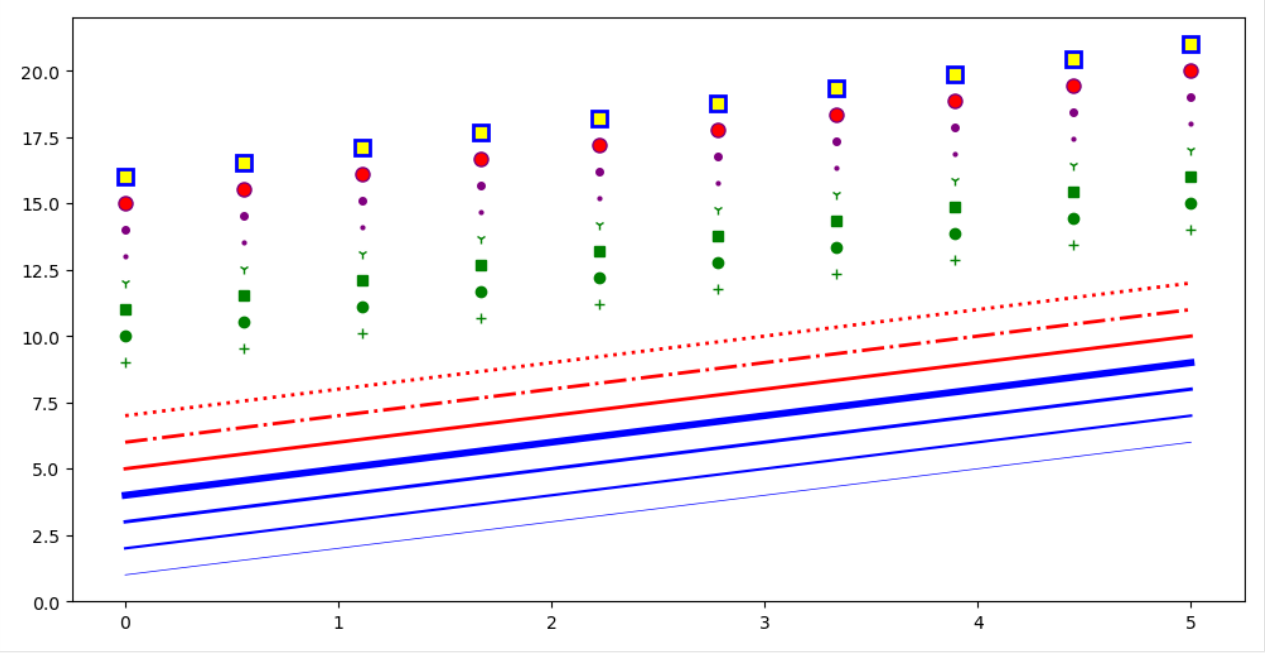
We can also use other keywords to enter various other information in `plt.plot`. To change the **line width**, we can use the `linewidth` or `lw` keyword argument. The **line style** can be selected using the `linestyle` or `ls` keyword arguments. Similarly, the **marker** type and **marker size** can be selected using the `marker` and `markersize` keyword arguments.

[7]:
```
plt.figure(figsize=(12,6))

plt.plot(x, x+1, color="blue", linewidth=0.5)
plt.plot(x, x+2, color="blue", linewidth=1.5)
plt.plot(x, x+3, color="blue", linewidth=2.0)
plt.plot(x, x+4, color="blue", linewidth=4.0)

# possible linestyle options ‘-‘, ‘--’, ‘-.’, ‘:’
plt.plot(x, x+5, color="red", lw=2, linestyle='-')
plt.plot(x, x+6, color="red", lw=2, ls='-.')
plt.plot(x, x+7, color="red", lw=2, ls=':')

# possible marker symbols: marker = '+', 'o', '*', 's', ',', '.', '1', '2', '3', '4', ...
plt.plot(x, x+9, color="green", ls='', marker='+')
plt.plot(x, x+10, color="green", ls='', marker='o')
plt.plot(x, x+11, color="green", ls='', marker='s')
plt.plot(x, x+12, color="green", ls='', marker='1')

# marker size and color
plt.plot(x, x+13, color="purple", ls='', marker='o', markersize=2)
```

(continues on next page)

(continued from previous page)

```
plt.plot(x, x+14, color="purple", ls='', marker='o', markersize=4)
plt.plot(x, x+15, color="purple", ls='', marker='o', markersize=8, markerfacecolor="red")
plt.plot(x, x+16, color="purple", ls='', marker='s', markersize=8,
        markerfacecolor="yellow", markeredgewidth=2, markeredgecolor="blue")
plt.show()
```



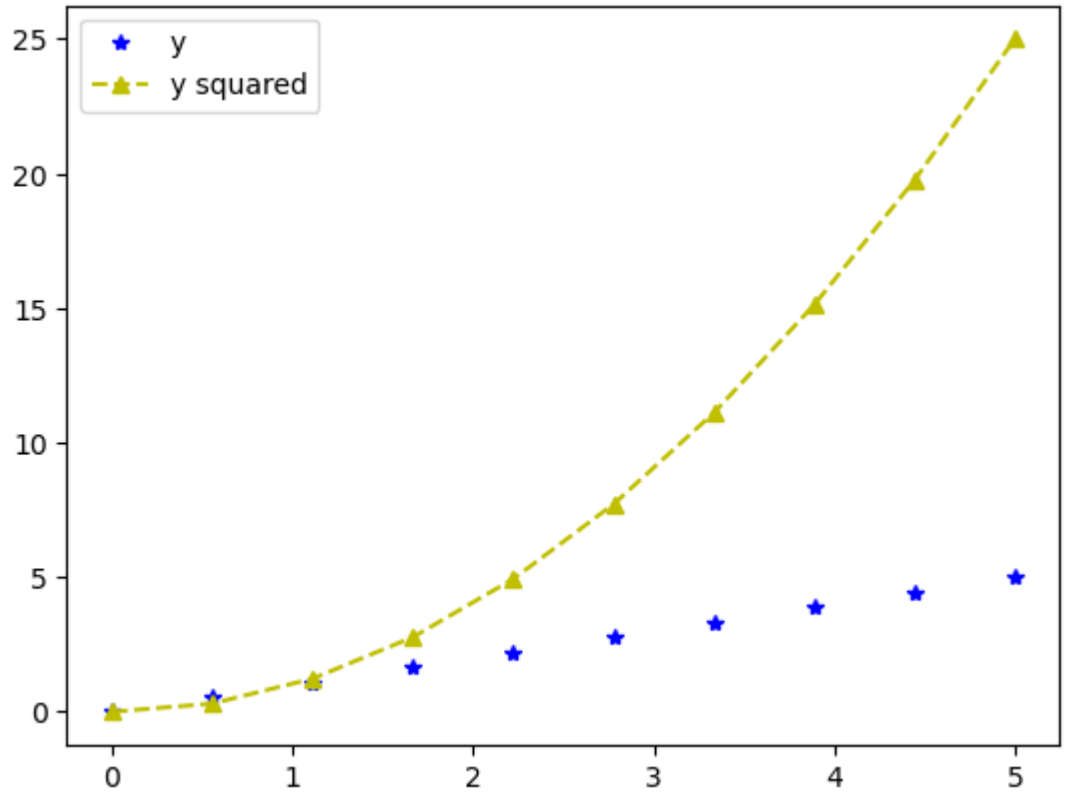
Multiple Lines in a Plot

We can plot multiple lines within a single plot, by calling `plt.plot` (or any other plotting function) multiple times. Below, we used two `plt.plot` functions to plot the squared values in the same figure.

Legend

Importantly, we can include a **legend** showing what each line represents using `plt.legend`. For the legend we pass a list or tuple of legend strings for the previously defined lines or curves. For the legend to be displayed correctly, the order of the labels `['y', 'y squared']` needs to match the order of the plotting calls. We can also specify the preferred location of the legend with the keyword `loc`.

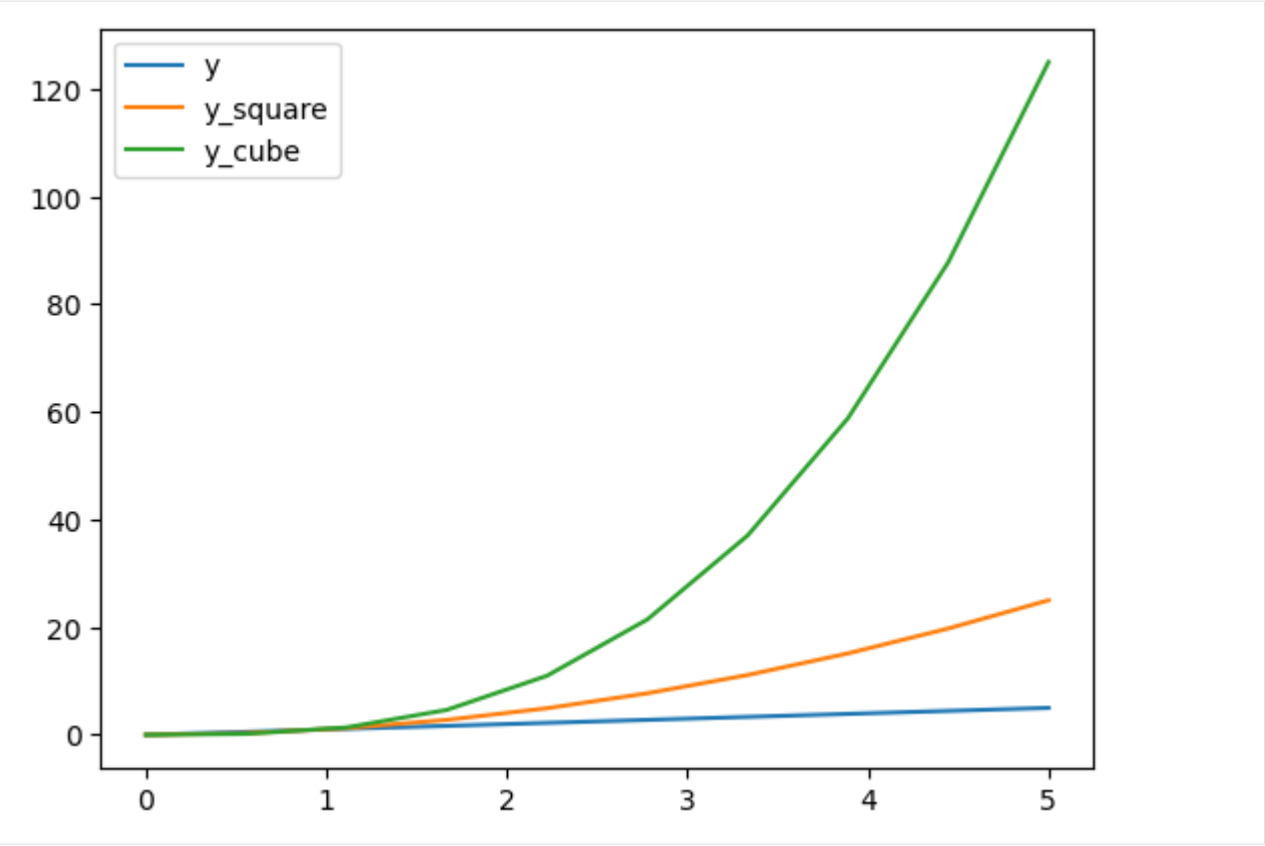
[8]:
```
plt.plot(x, x, '*b')  # only plot markers (*) in blue
plt.plot(x, x**2, '^--y')  # plot both markers (^) and a dashed line (--) in yellow

# Note that the plt.legend function call should come *after* the plotting calls
# and you should give it a *list* with strings
plt.legend(['y', 'y squared'], loc='upper left')
plt.show()
```



Another method for adding a **legend** is to use the `label='label_text'` keyword argument when plots are added to the figure, and then use the `legend` method without arguments, as shown in the code below. The advantage of this method is that if lines/curves are added or removed from the figure, the legend is automatically updated accordingly.

One more thing to notice in the plot below is that when we plot multiple items in the same figure, if we don't specify the format and color, Matplotlib will automatically choose a different color for the different items (first one is blue, second one is orange, third one is green, etc.).

[9]:
```
plt.plot(x, x, label='y')
plt.plot(x, x**2, label='y_square')
plt.plot(x, x**3, label='y_cube')
plt.legend()
plt.show()
```



Anatomy of a Figure

A general anatomy of a figure is shown below depicting the different items and properties of a figure.

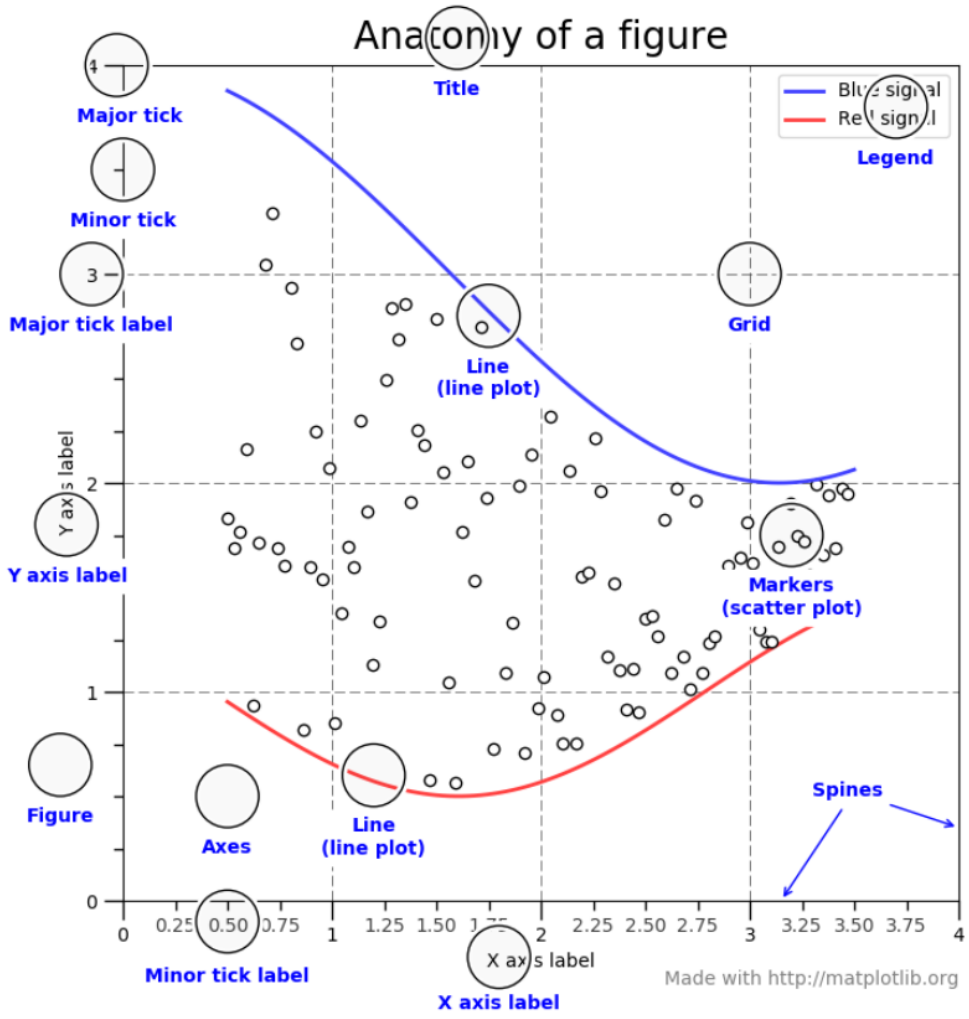
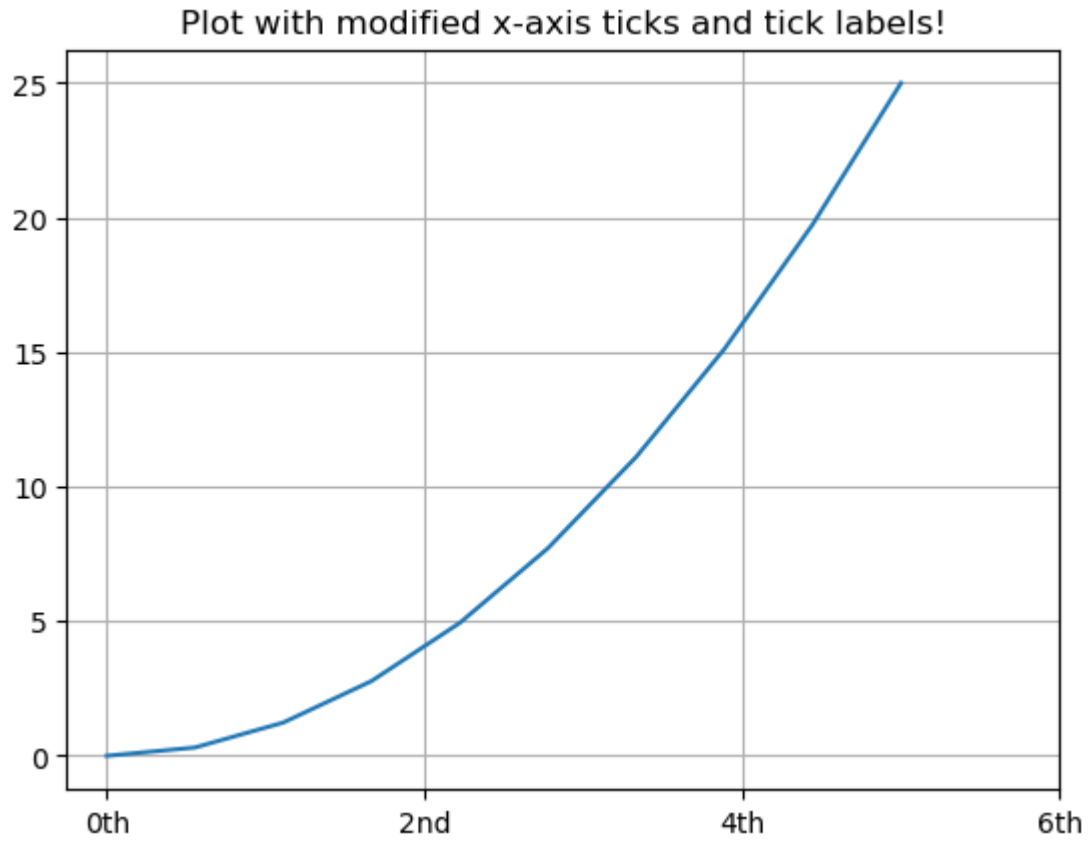


Figure source: Reference [3].

Figure Title, Ticks, Tick Labels

Matplotlib provides various functions to modify figures, and for instance, if we wish we can add a **title** with `plt.title` or change the default **ticks** and **tick labels** using `plt.xticks` (for the x-axis ticks/tick labels) and `plt.yticks` (for the y-axis ticks/tick labels). An example follows below, which shows how we can set custom tick labels for each tick position.

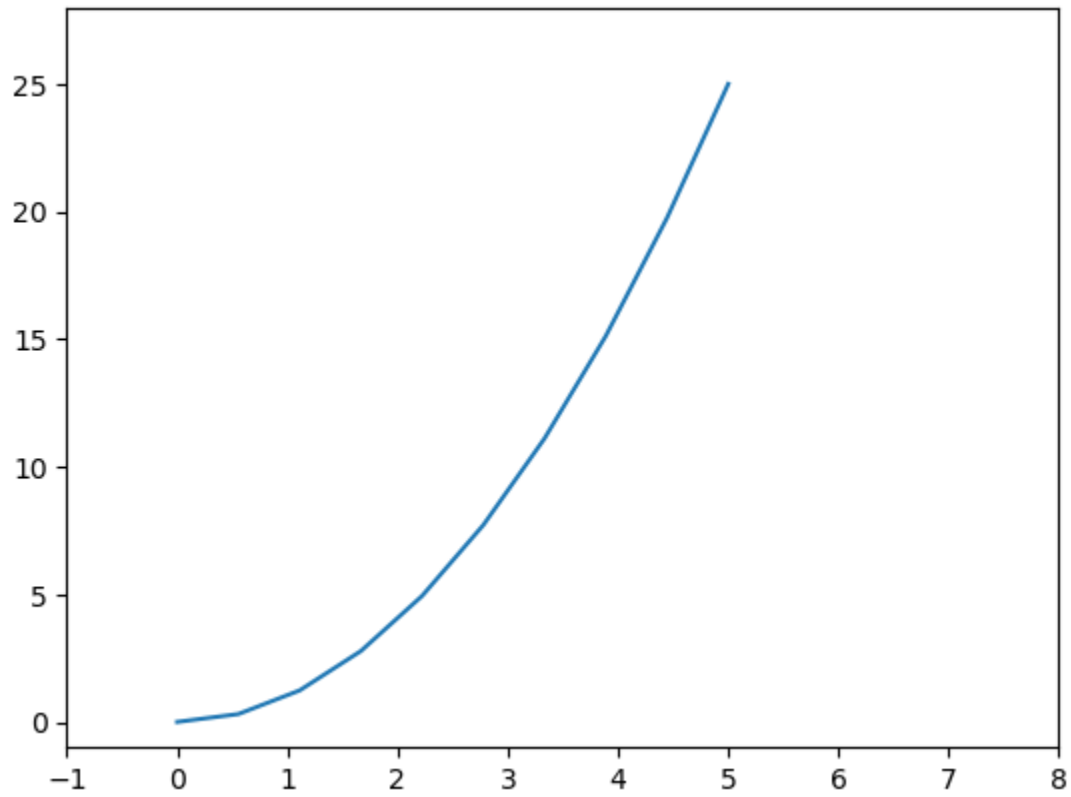
[10]:
```
plt.title("Plot with modified x-axis ticks and tick labels!", fontsize=12)
plt.plot(x, x**2)
plt.xticks([0, 2, 4, 6], ['0th', '2nd', '4th', '6th'])  # a list of x-axis positions,␣
↪followed by a list of x-tick labels
plt.grid(True)
plt.show()
```



Range

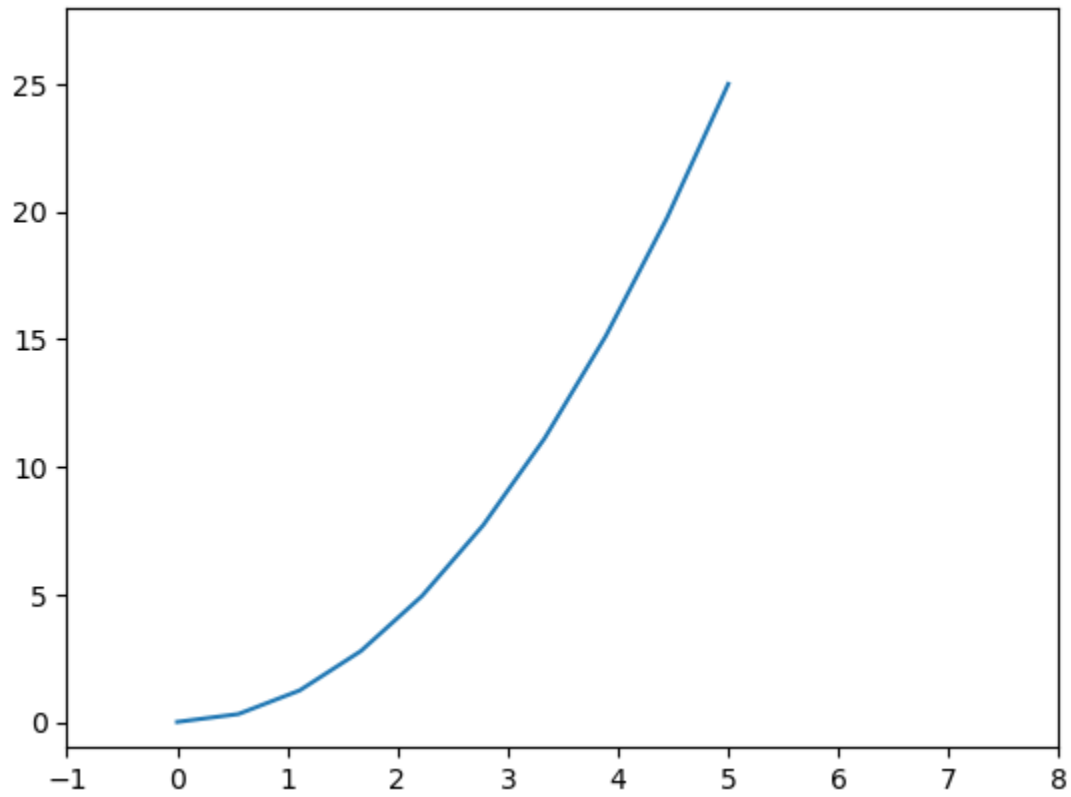
Also, we can control the **range** of the axes by the functions `plt.xlim` and `plt.ylim`.

[11]:
```
plt.plot(x, x**2)
plt.xlim(-1, 8)
plt.ylim(-1, 28)
plt.show()
```



Similarly, we can control the range with the `plt.axis()` function which takes a list of arguments `[xmin, xmax, ymin, ymax]`.

[12]:
```
plt.plot(x, x**2)
plt.axis([-1, 8, -1, 28])
plt.show()
```



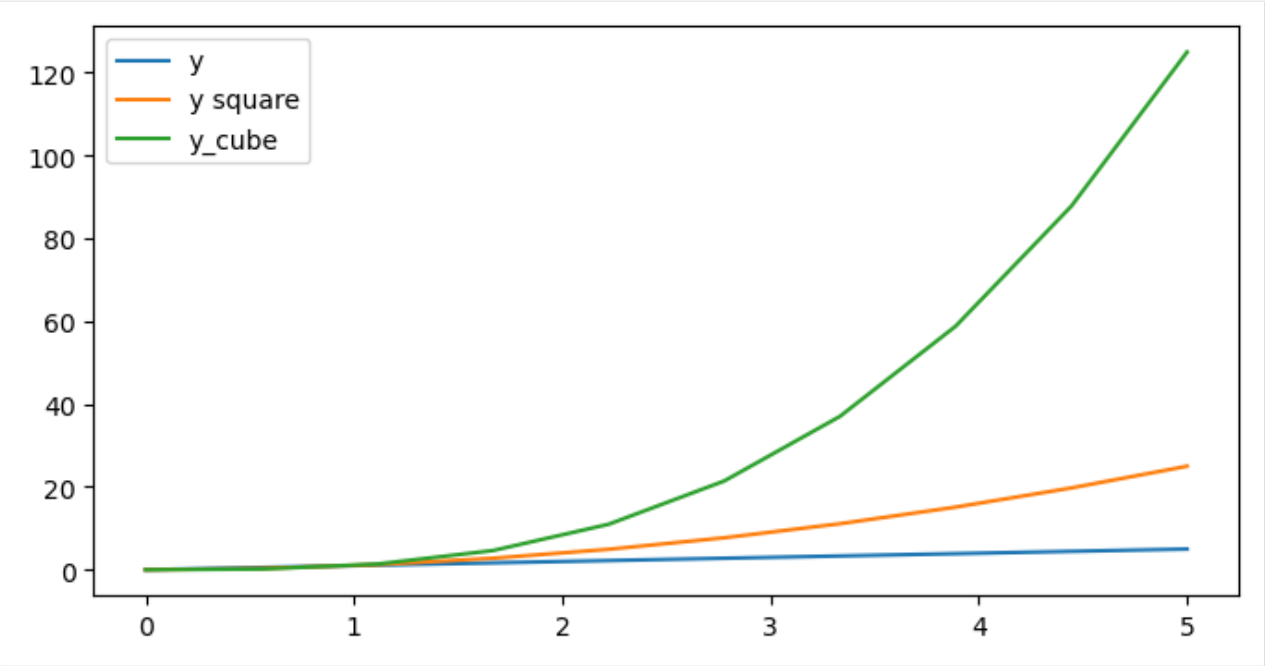
7.9.3 9.3 Figure Size, Aspect Ratio, and DPI

Matplotlib allows to specify the aspect ratio, DPI (dots-per-inch resolution), and figure size by calling the `plt.figure` function, and using the `figsize` and `dpi` keyword arguments. `figsize` is a tuple of the width and height of the figure in inches, and `dpi` is the dots-per-inch (pixel per inch) resolution. The larger the DPI, the larger the resolution of the figure.

If we don't include any arguments in `plt.figure`, Matplotlib creates a new figure with default size of 6.4 x 4.8 inches, and a default DPI of 100.

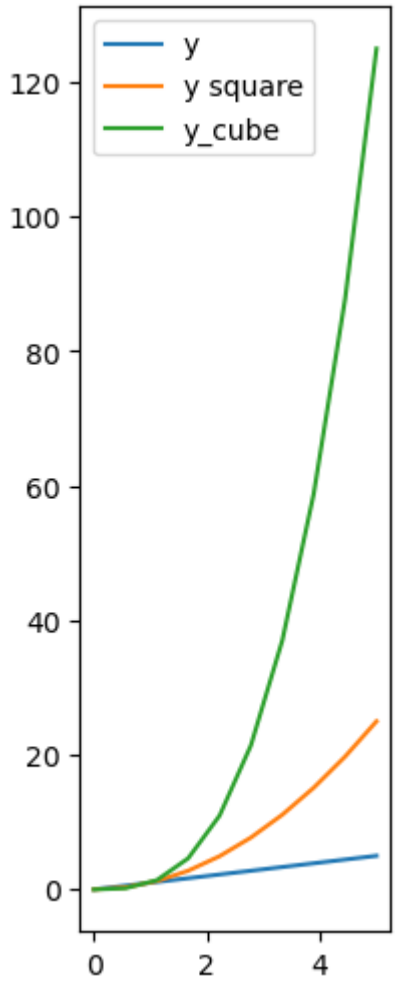
The following example is used to create an 8x4 inches figure (width x height) with 100 DPI, that is, the figure is 800x400 pixels.

[13]:
```
plt.figure(figsize=(8,4), dpi=100)
plt.plot(x, x)
plt.plot(x, x**2)
plt.plot(x, x**3)
plt.legend(['y', 'y square', 'y_cube'])
plt.show()
```



Similarly, the following figure is 2x6 inches (width x height) with 100 DPI, i.e., it is 200x600 pixels.

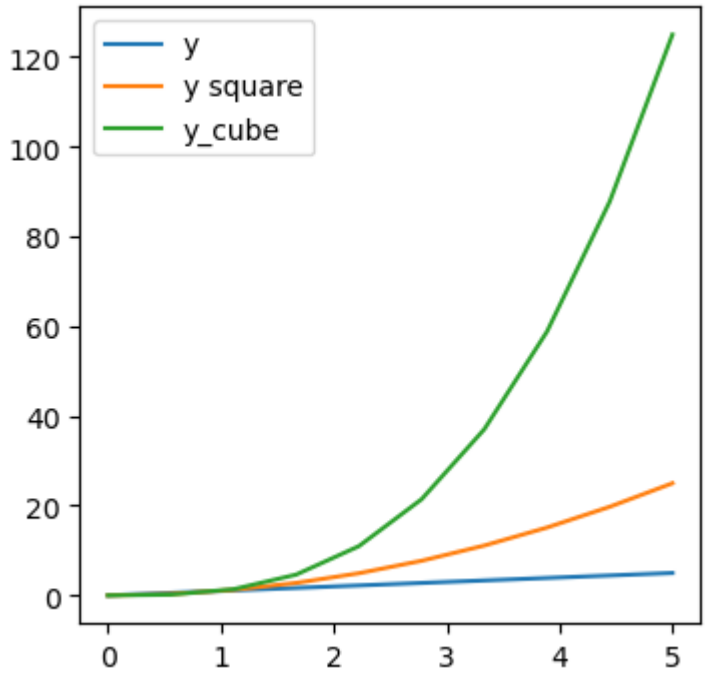
[14]:
```
plt.figure(figsize=(2,6), dpi=100)
plt.plot(x, x)
plt.plot(x, x**2)
plt.plot(x, x**3)
plt.legend(['y', 'y square', 'y_cube'])
plt.show()
```



7.9.4 9.4 Saving Figures

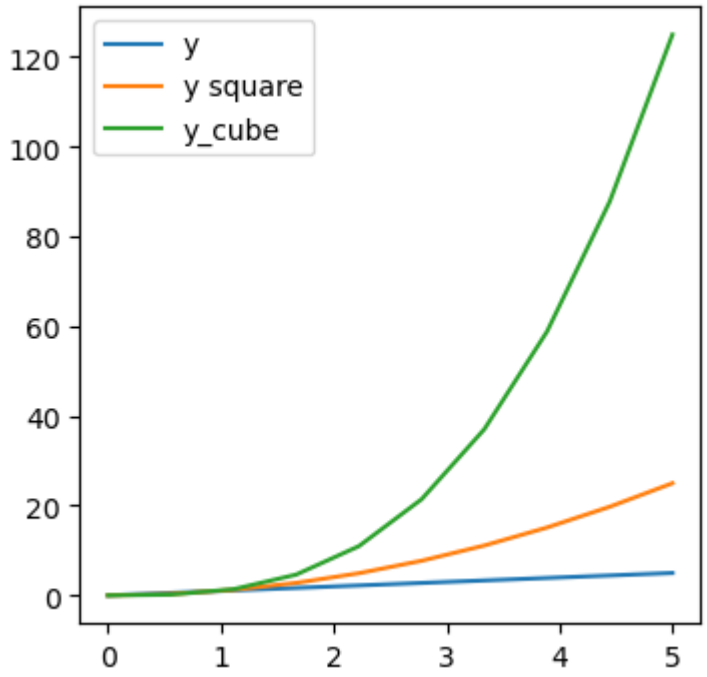
To save a figure to a file we can use the `plt.savefig` function with the name of the figure as argument. By using the `.png` file suffix in the name `"figure_1.png"` below, we chose to save the figure in the PNG file format.

[15]:
```
plt.figure(figsize=(4,4))
plt.plot(x, x)
plt.plot(x, x**2)
plt.plot(x, x**3)
plt.legend(['y', 'y square', 'y_cube'])
# plt.show() - this line is not needed when saving a figure
plt.savefig("figure_1.png")
```



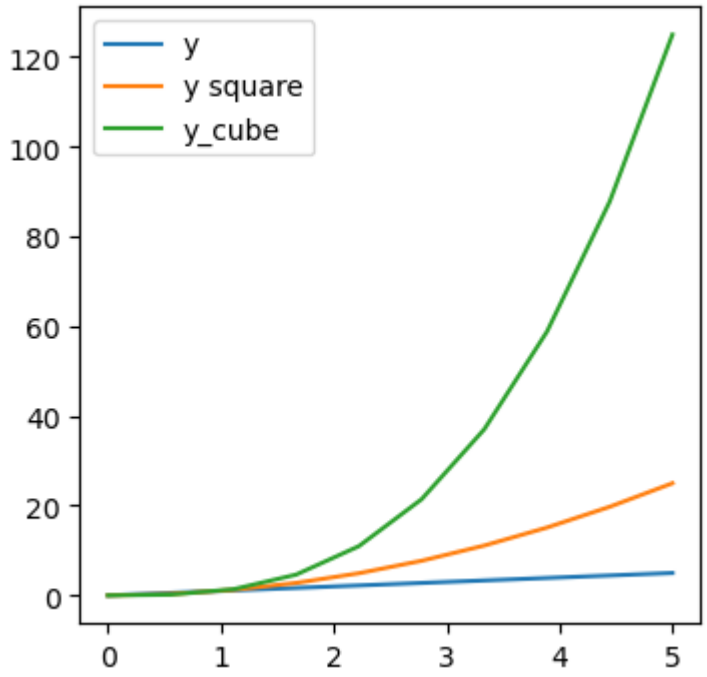
We can also optionally specify the DPI of the figure in `plt.savefig()`, as well as choose another output format (e.g., JPG).

[16]:
```
plt.figure(figsize=(4,4))
plt.plot(x, x)
plt.plot(x, x**2)
plt.plot(x, x**3)
plt.legend(['y', 'y square', 'y_cube'])
plt.savefig("figure_2.jpg", dpi=200)
```



Matplotlib can generate high-quality figures in various formats, including PNG, JPG, EPS, SVG, PGF, and PDF. The file format can be conveniently specified via the file suffix (.eps, .svg, .jpg, .png, .pdf, .tiff). Using a vector graphics format (.eps, .svg, .pdf) is often recommended, because it usually results in smaller file sizes than bitmap graphics (.jpg, .png, .bmp, .tiff) and does not have a limited resolution.

[17]:
```
plt.figure(figsize=(4,4))
plt.plot(x, x)
plt.plot(x, x**2)
plt.plot(x, x**3)
plt.legend(['y', 'y square', 'y_cube'])
plt.savefig("figure_3.pdf", dpi=200)
```

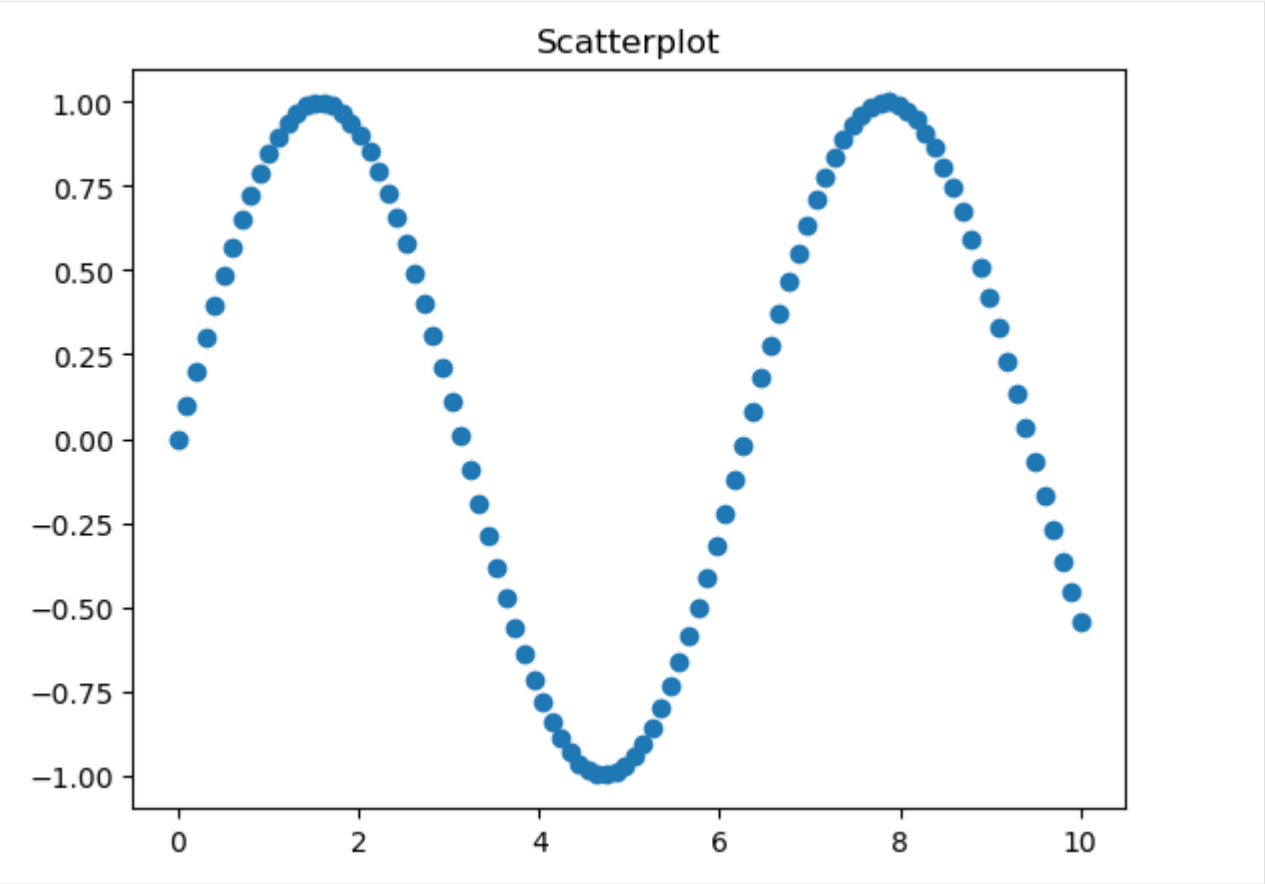


7.9.5 9.5 Other Plotting Functions

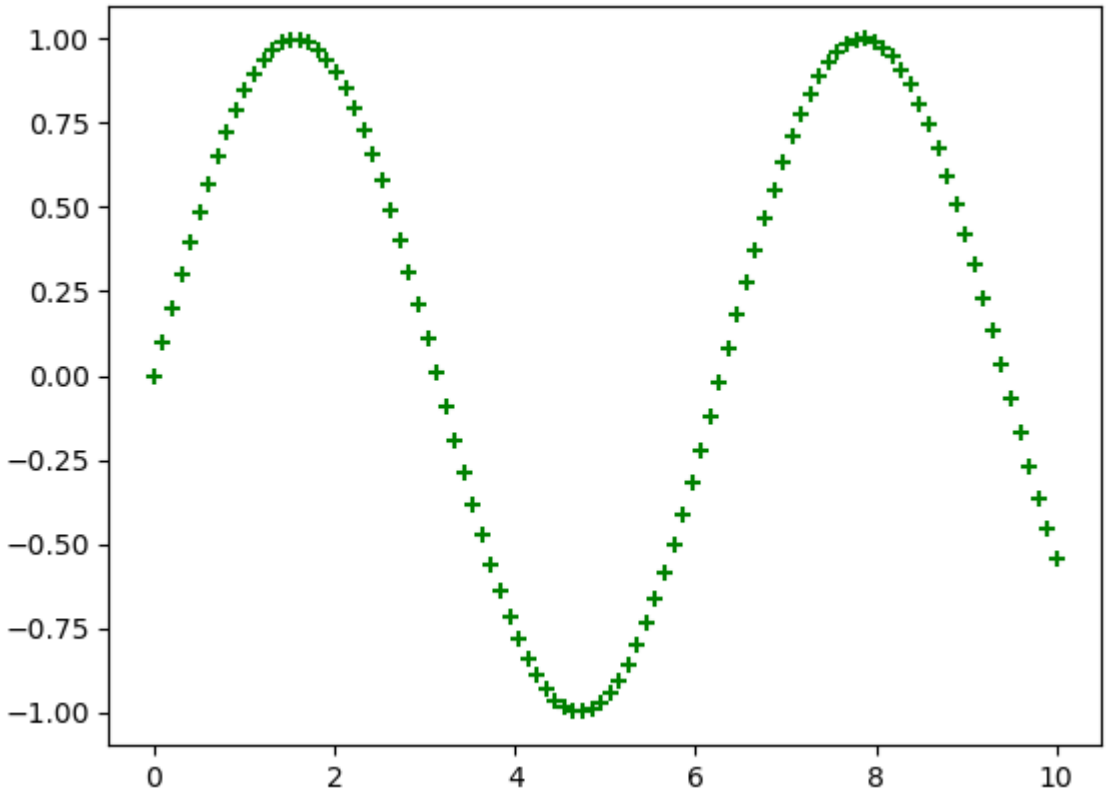
Scatter Plots

There are many different plotting functions in Matplotlib besides `plt.plot`. Scatterplots can be created using `plt.scatter`.

[18]:
```
x2 = np.linspace(0, 10, 100)
plt.scatter(x2, np.sin(x2))  # This is equivalent to plt.plot(x, y, 'o') !
plt.title("Scatterplot", fontsize=12)
plt.show()
```



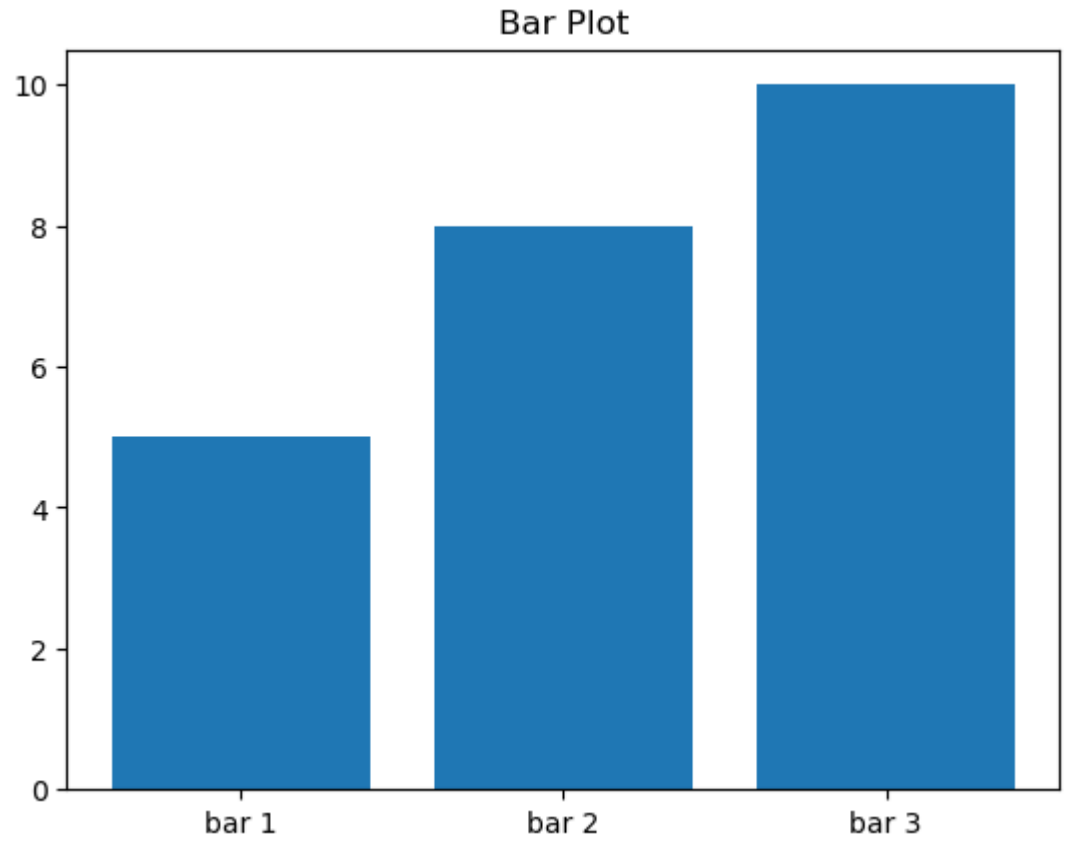
[19]:
```
plt.scatter(x2, np.sin(x2), color='green', marker='+')
plt.show()
```



Bar Plots

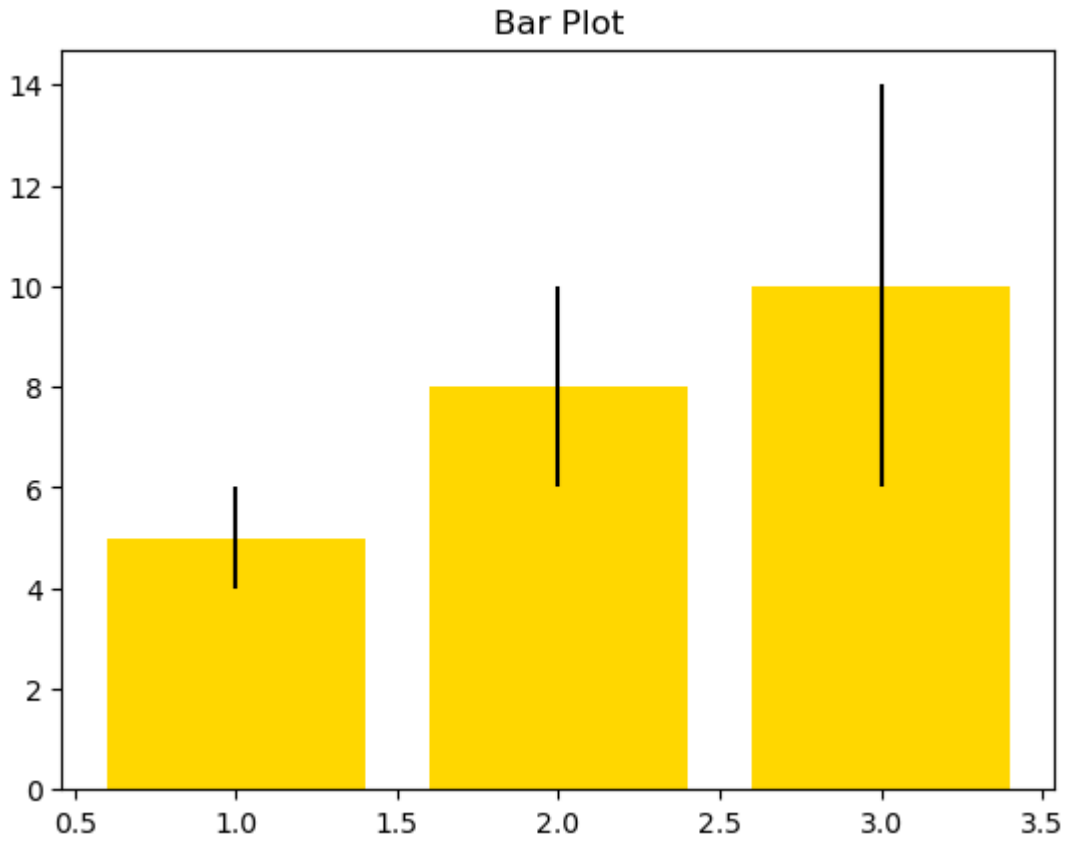
Bar graphs are created using `plt.bar`. We pass in a list of values or strings for the x-axis, and a list of heights for the bars on the y-axis. We can optionally set a color for the bars (and if we don't, the default color is blue).

[20]:
```
# First argument determines the location of the bars on the x-axis
# and the second argument determines the height of the bars
bar_labels = ['bar 1', 'bar 2', 'bar 3']
means = [5, 8, 10]
plt.bar(bar_labels, means)
plt.title("Bar Plot", fontsize=12)
plt.show()
```



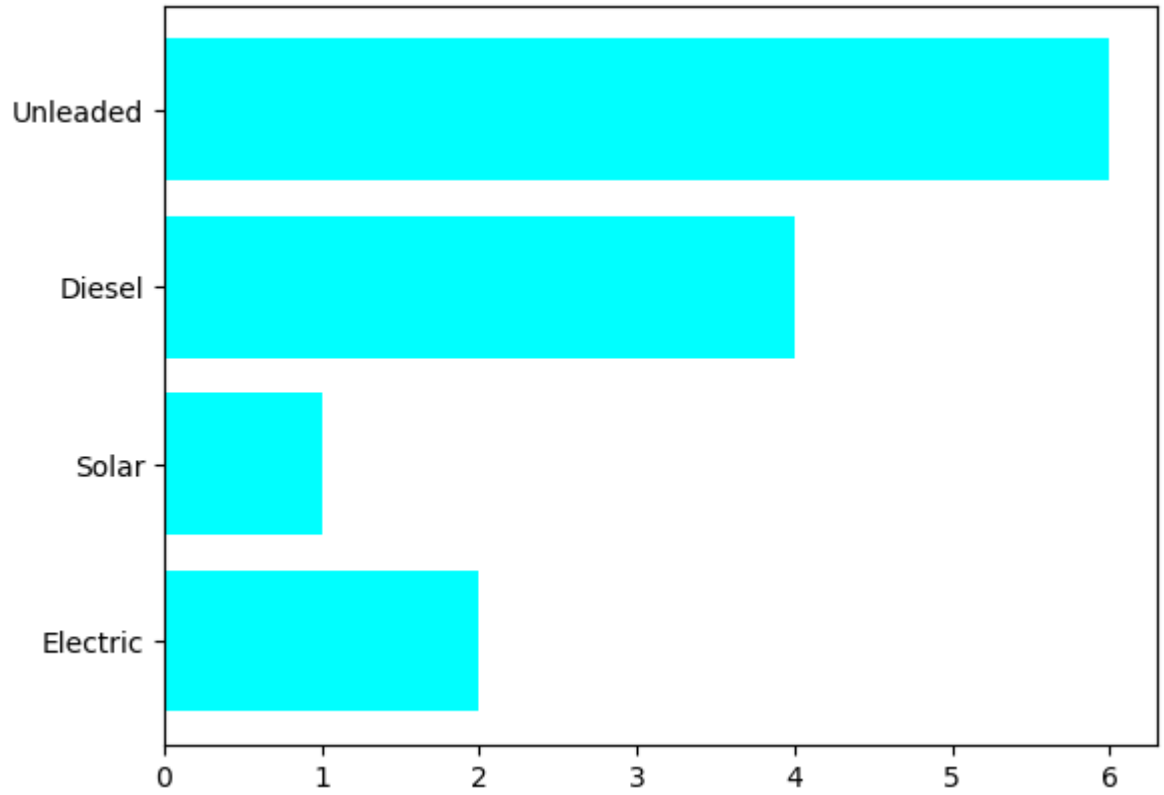
We can also add error bars to the bar plot using the `yerr` (y-error) keyword in `plt.bar`.

[21]:
```
bar_labels = [1, 2, 3]
means = [5, 8, 10]
variances = [1, 2, 4]
plt.bar(bar_labels, means, yerr=variances, color='gold')
plt.title("Bar Plot", fontsize=12)
plt.show()
```



And, we can create horizontal bar plots using `plt.barh`. In this case, the first argument is a list of positions on the y-axis, and the second argument is a list of values of the bars along the x-axis.

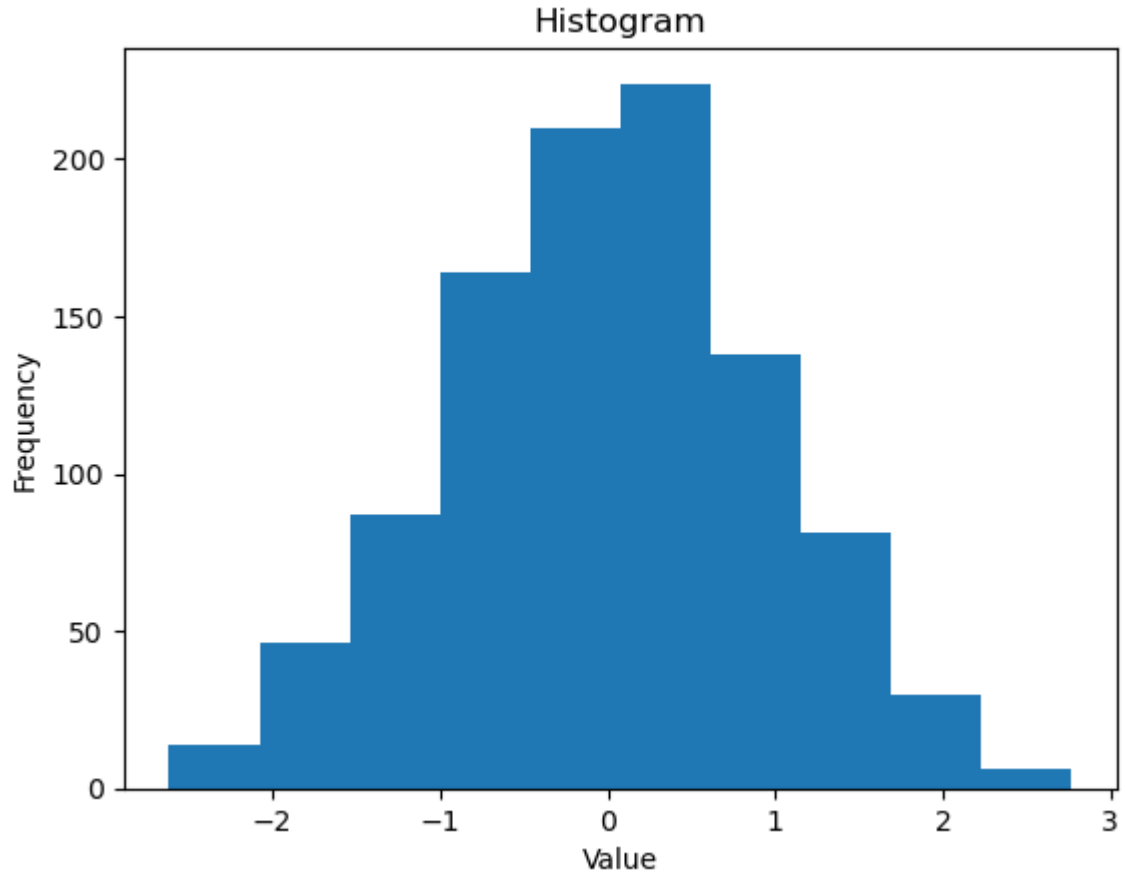
[22]:
```
labels = ['Electric', 'Solar', 'Diesel', 'Unleaded']
values = [2, 1, 4, 6]
plt.barh(labels, values, color='cyan')
plt.show()
```



Histograms

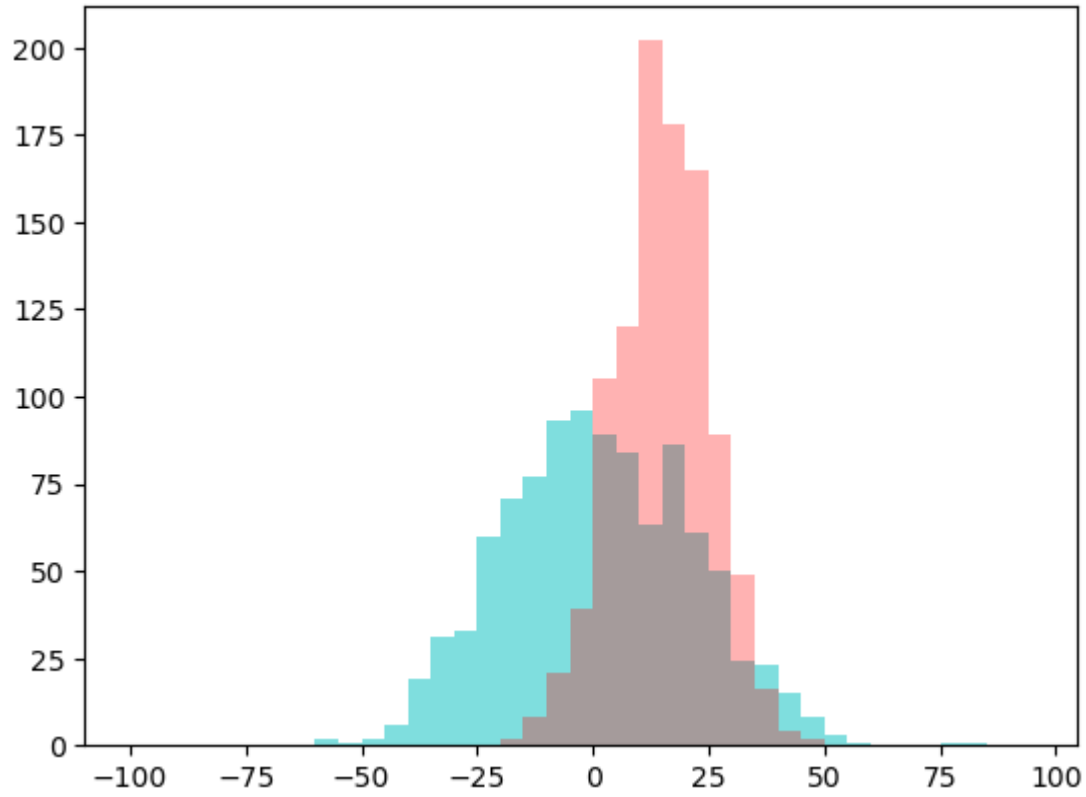
The function `plt.hist` is used for this purpose.

[23]:
```
# Let's generate some random data from normal distribution
random_norm = np.random.randn(1000)

plt.title("Histogram", fontsize=12)
plt.hist(random_norm)
plt.xlabel("Value", fontsize=10)
plt.ylabel("Frequency", fontsize=10)
plt.show()
```



Histograms of 2 normal distributions with different mean and variance values are shown below. Note that the `alpha` keyword controls the transparency level of the histograms, where 0.3 corresponds to 30% transparency. The `bins` keyword controls the number of bins to divide the values in the distributions.

[24]:
```
# Mean 0 and variance 20
random_norm1 = 0 + 20*np.random.randn(1000)
# Mean 15 and variance 10
random_norm2 = 15 + 10*np.random.randn(1000)

# fixed bin size
bins = np.arange(-100, 100, 5)

plt.hist(random_norm1, bins=bins, facecolor='c', alpha=0.5)
plt.hist(random_norm2, bins=bins, facecolor='r', alpha=0.3)
plt.show()
```

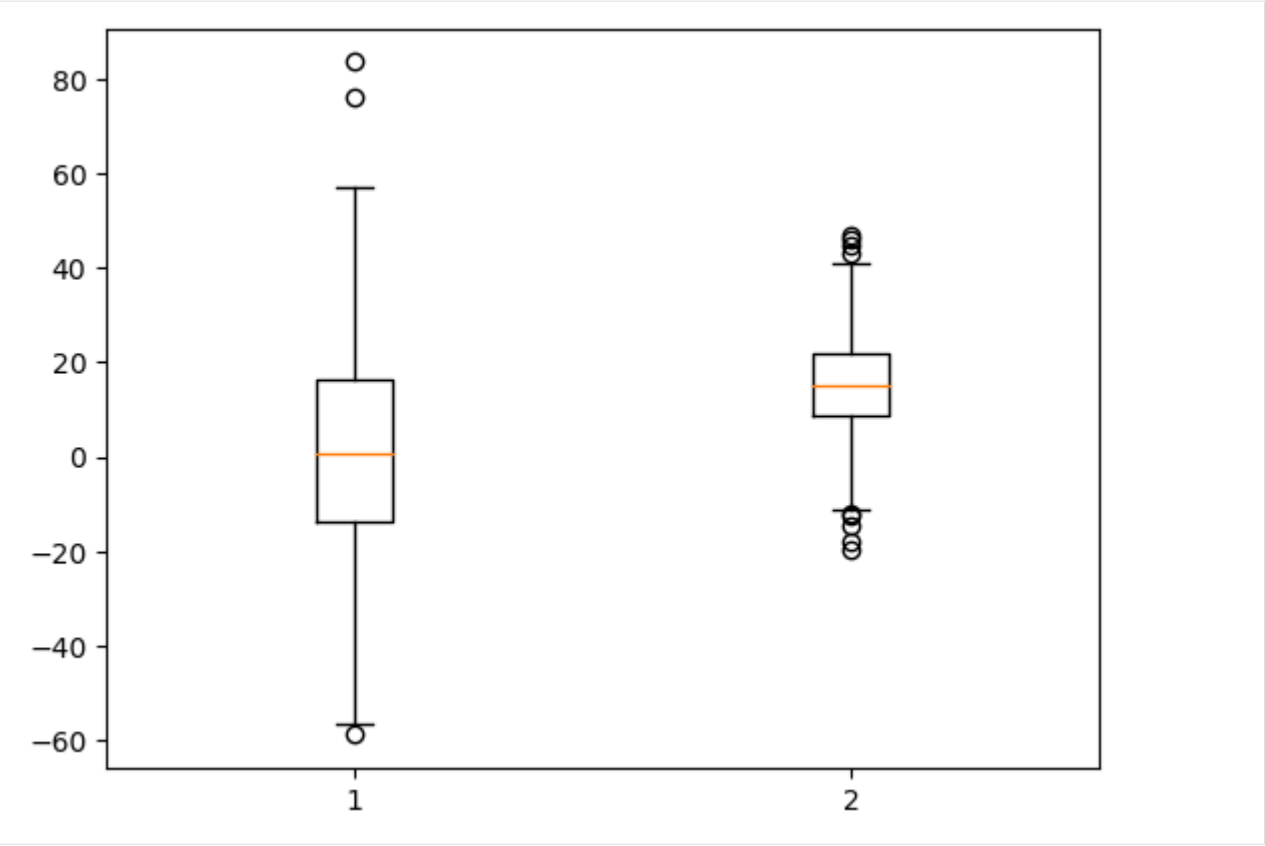


Boxplots

The function `plt.boxplot` can be used to plot the distribution of data with a boxplot, showing the median of the data (the vertical line in the middle), the interquartile ranges (i.e., the ends of the boxes represent 25 and 75 percentiles), and the minimum and maximum values of the data (that is, the far end of the lines extending from the boxes, referred to as "whiskers"). The values outside of the whiskers are outliers in the data.

Boxplots for the two random distributions from the previous example are shown below.

[25]:
```
plt.boxplot([random_norm1, random_norm2])
plt.show()
```

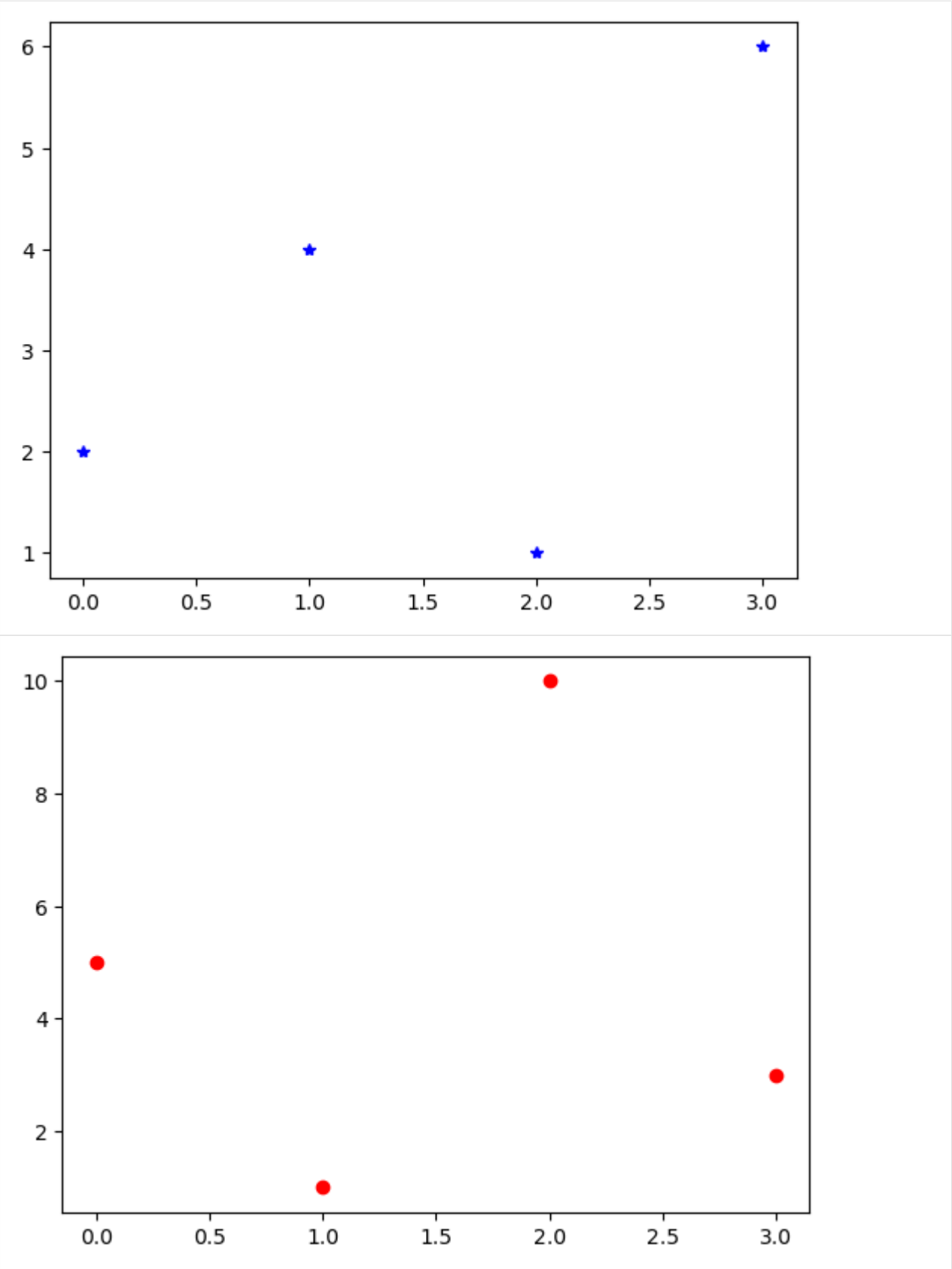


7.9.6 9.6 Multiple Plots in Figures

Matplotlib allows creating several plots in the same figure. This allows working with multiple datasets at once. There are several different ways to do this. The simplest way is to create a new figure to add the second `plt.plot` function.

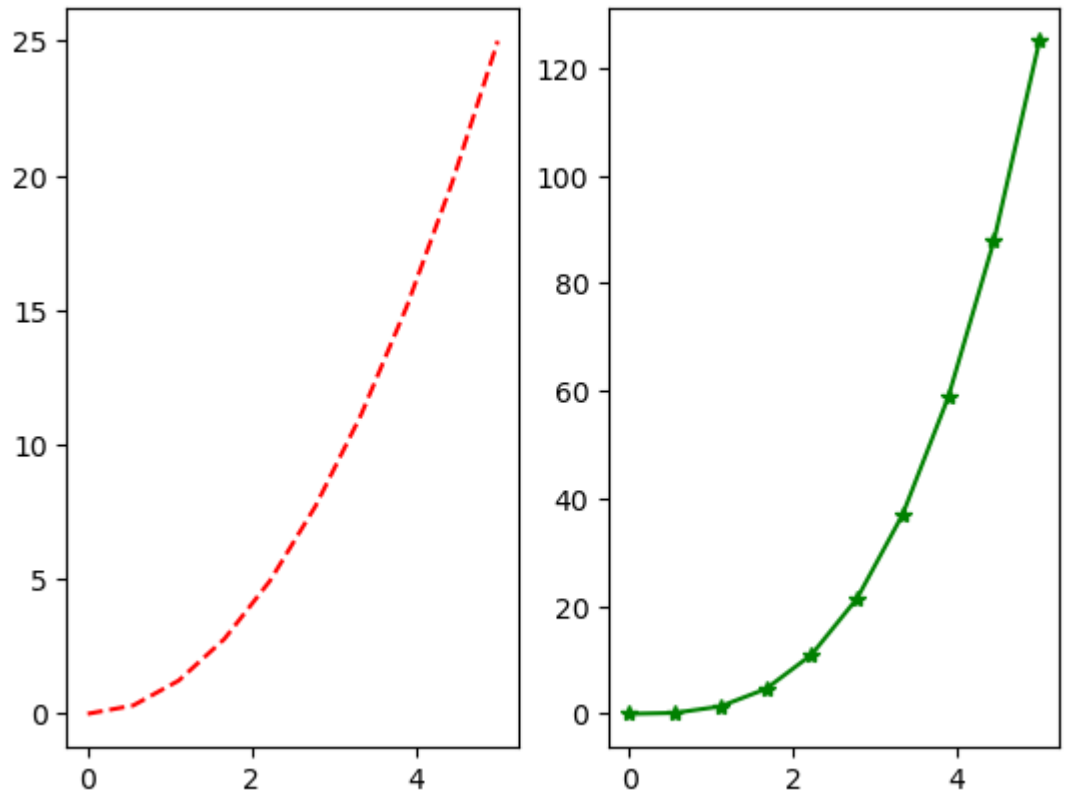
Here we created two line plots. When we called `plt.figure(2)`, it created a top-level container for the `plt.plot` function that follows after it. Thus, the first plot is added to figure one (which is created automatically when `plt.plot` is called), and the second plot is added to figure 2. When we call `show()` at the end, Matplotlib will open two windows with each graph shown separately.

[26]:
```
numbers1 = [2, 4, 1, 6]
numbers2 = [5, 1, 10, 3]

plt.plot(numbers1, '*b')

second_plot = plt.figure(2)
plt.plot(numbers2, 'or')
plt.show()
```


6
5
4
3
2
1
0.0
0.5
1.0
1.5
2.0
2.5
3.0
10
8
6
4
2
0.0
0.5
1.0
1.5
2.0
2.5
3.0

Matplotlib also supports adding two or more plots to a single figure, by using the function `subplot`. The arguments in `subplot` define the number of rows, the number of columns, and the number of each subplot.

[27]:
```
plt.subplot(1,2,1)
plt.plot(x, x**2, 'r--')
plt.subplot(1,2,2)
plt.plot(x, x**3, 'g*-');
```



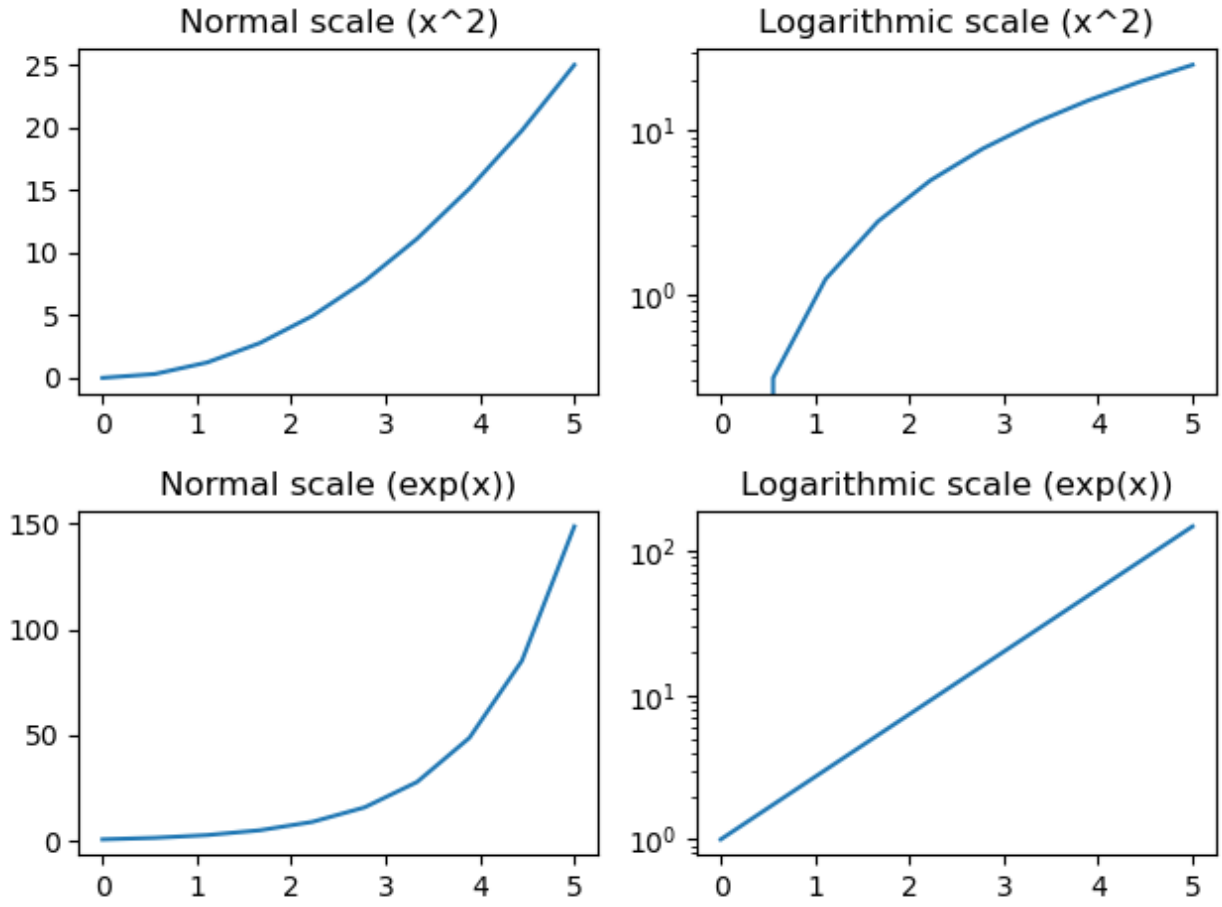
In the following figure, the right plots use a logarithmic scale with the function `plt.yscale("log")`. One more thing to note is that the arguments in `plt.subplots` do not need to be separated by commas.

[28]:
```
plt.subplot(221)
plt.plot(x, x**2)
plt.title("Normal scale (x^2)")

plt.subplot(222)
plt.plot(x, x**2)
plt.yscale("log")
plt.title("Logarithmic scale (x^2)")

plt.subplot(223)
plt.plot(x, np.exp(x))
plt.title("Normal scale (exp(x))")

plt.subplot(224)
plt.plot(x, np.exp(x))
plt.yscale("log")
```

(continues on next page)

(continued from previous page)

```
plt.title("Logarithmic scale (exp(x))")

plt.tight_layout()
```



7.9.7 9.7 The Object-oriented Approach

The state-based plotting approach in Matplotlib is easy to use and pretty straightforward, however for creating more complex visualizations, the alternative object-oriented approach provides advantages.

The main idea of the **object-oriented approach** in Matplotlib is to create objects to which we can apply methods. In this approach, each Matplotlib plot consists of a Figure object and one or more Axes objects. Essentially, the Figure object represents the entire canvas that defines the figure. The Axes objects contain the actual visualizations that we want to include in the Figure. This is shown below. However, note that an Axes object is different than the axes (x-axis and y-axis) of a plot. And, there may be one or multiple Axes objects within a given Figure (e.g., two line plots next to each other).

The real advantage of this approach becomes apparent when more than one figure is created, or when a figure contains more than one subplot.

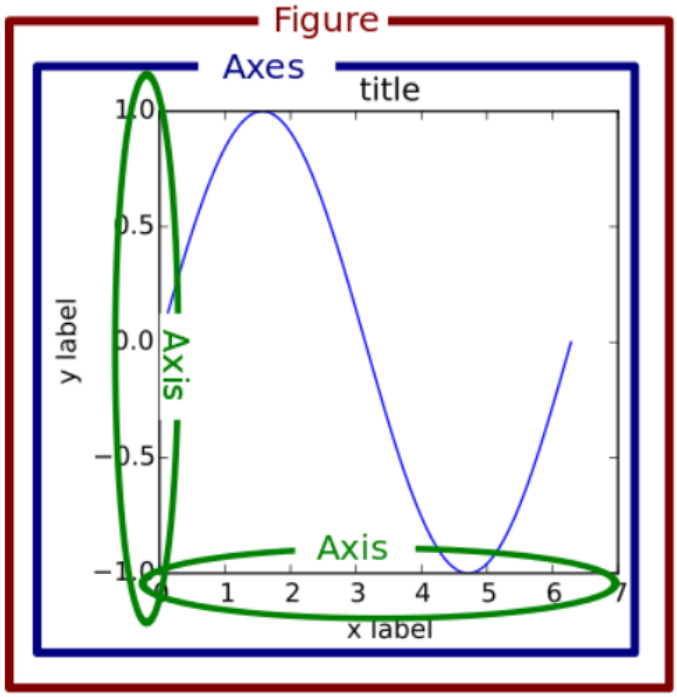
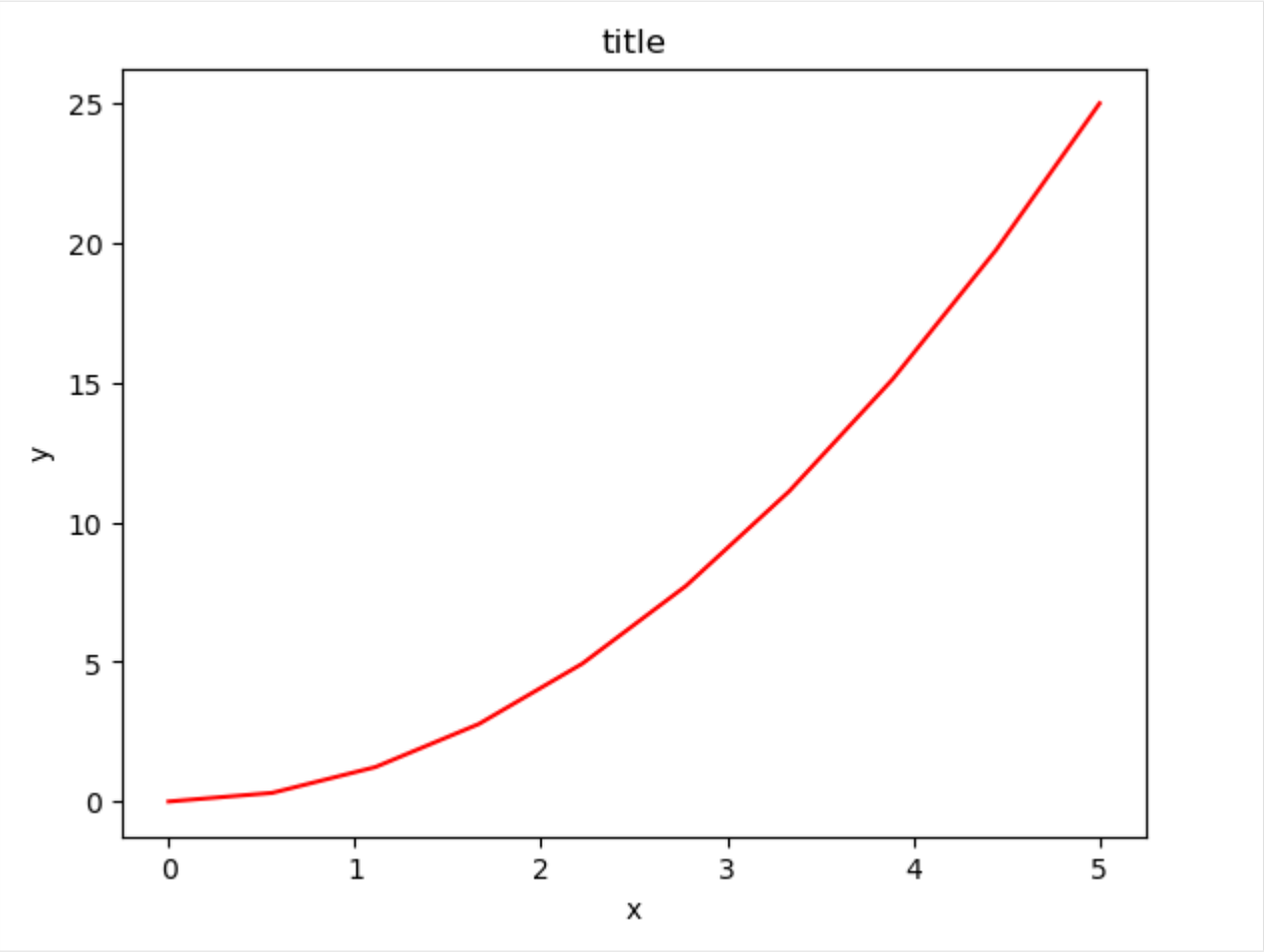


Figure source: Reference [3].

To use the object-oriented approach, we create a new Figure object that defines the canvas to draw on by using `plt.figure()`. In the next cell, we passed the newly created Figure instance to the `fig` name. As we explained before, `plt.figure` can take optional arguments like figsize (width and height in inches) and dpi.

Next, from the Figure class instance `fig` we create a new Axes instance `axes` using the `add_axes` method. In the object-oriented approach, plotting is done through the methods of the newly created `axes` object, instead of the function `plot` (or `scatter`, `bar`) from the pyplot module. And, even in the object-oriented approach we need the function `plt.show` to render the figure.

[29]:
```
fig = plt.figure()

axes = fig.add_axes([0, 0, 0.8, 0.8]) # left, bottom, width, height (range 0 to 1)

axes.plot(x, x**2, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title')
plt.show()
```



Basically all functions from the state-based approach are available as methods in the object-oriented approach. Also, some pyplot functions are prefixed with `set_` in the object-oriented interface. For instance, `plt.title` or `plt.xlabel` functions used with the state-based approach, in the object-oriented approach are replaced with `axes.set_title` or `axes.set_xlabel`.

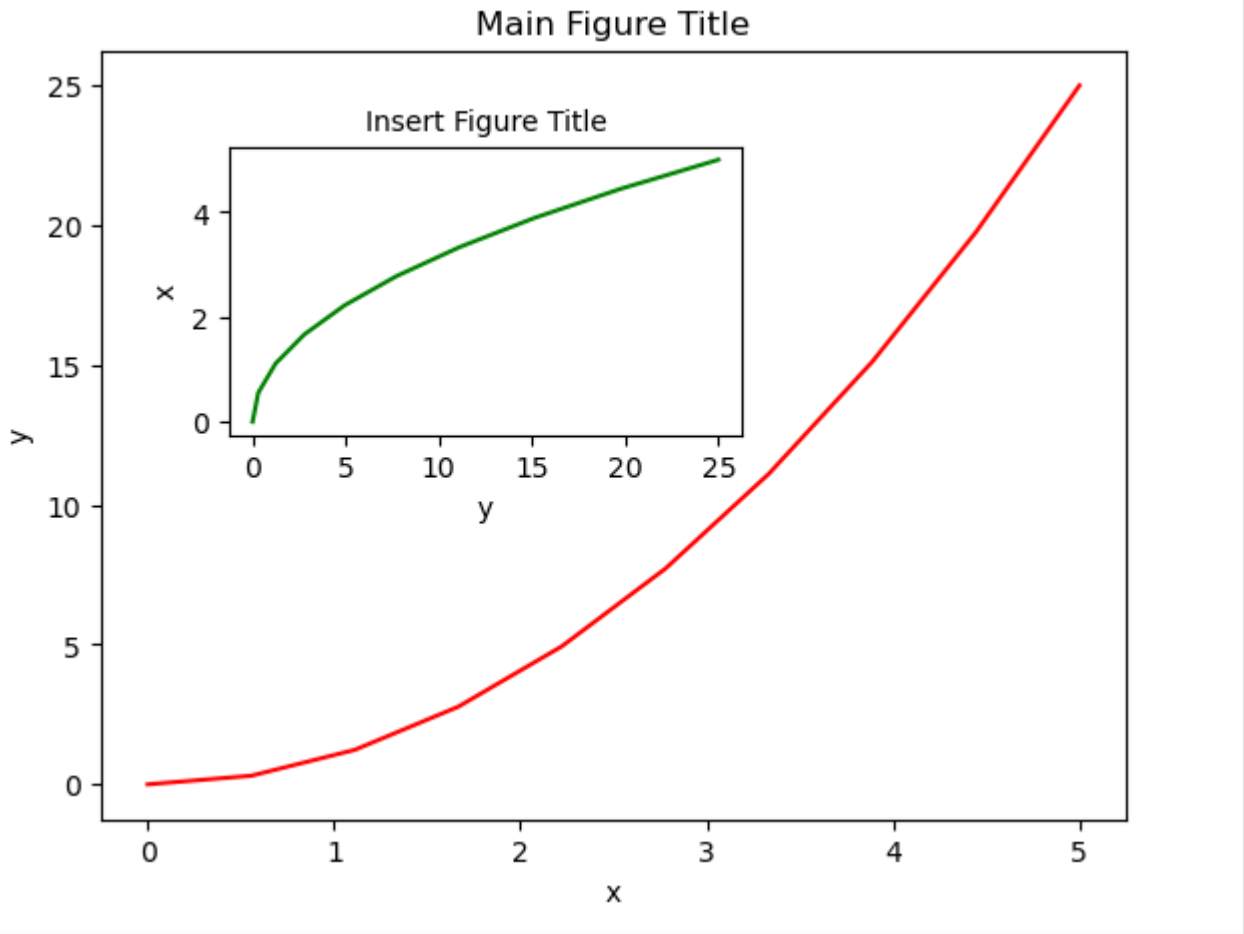
The advantage of the object-oriented approach is that it enables having full control of where the plot axes are placed, and we can easily add more than one axis to the figure.

[30]:
```
fig = plt.figure()

axes1 = fig.add_axes([0, 0, 0.8, 0.8]) # main axes
axes2 = fig.add_axes([0.1, 0.4, 0.4, 0.3]) # inset axes

# main figure
axes1.plot(x, x**2, 'r')
axes1.set_xlabel('x')
axes1.set_ylabel('y')
axes1.set_title('Main Figure Title', fontsize=12)

# insert
axes2.plot(x**2, x, 'g')
axes2.set_xlabel('y')
axes2.set_ylabel('x')
```

(continues on next page)

(continued from previous page)

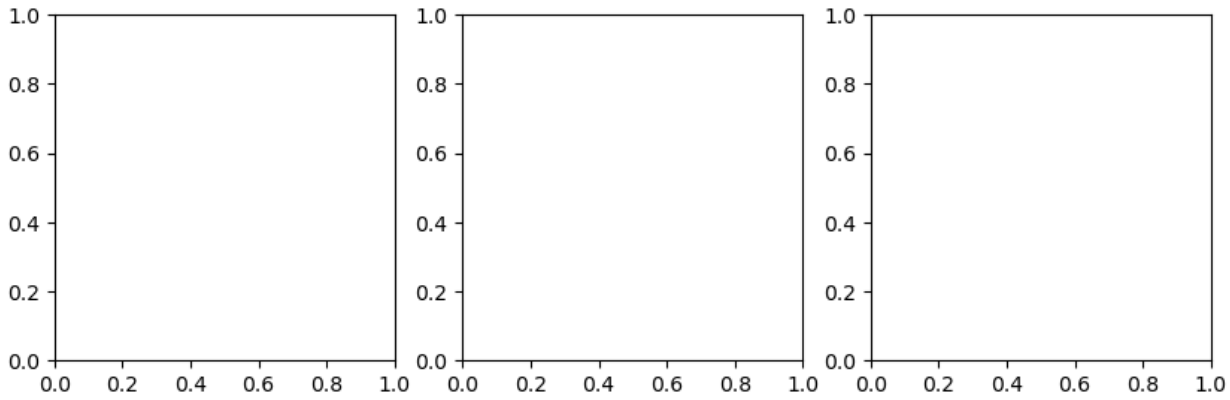
```
axes2.set_title('Insert Figure Title', fontsize=10)

fig.savefig("figure_4.jpg", dpi=200)
```



In the above example, we saved the figure using the method `fig.savefig`, instead of `plt.savefig` that we used in the state-based approach.

Instead of creating the Figure and Axes objects separately, we can use the function `plt.subplots` to create them both at the same time, as in the next example. As the name suggests, this function also allows to create multiple subplots where each subplot corresponds to one Axes object). The `plt.subplots` function accepts as arguments the number of rows and columns `nrows` and `ncols`, and optionally the figure size.

[31]:
```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(10, 3))
plt.show()
```



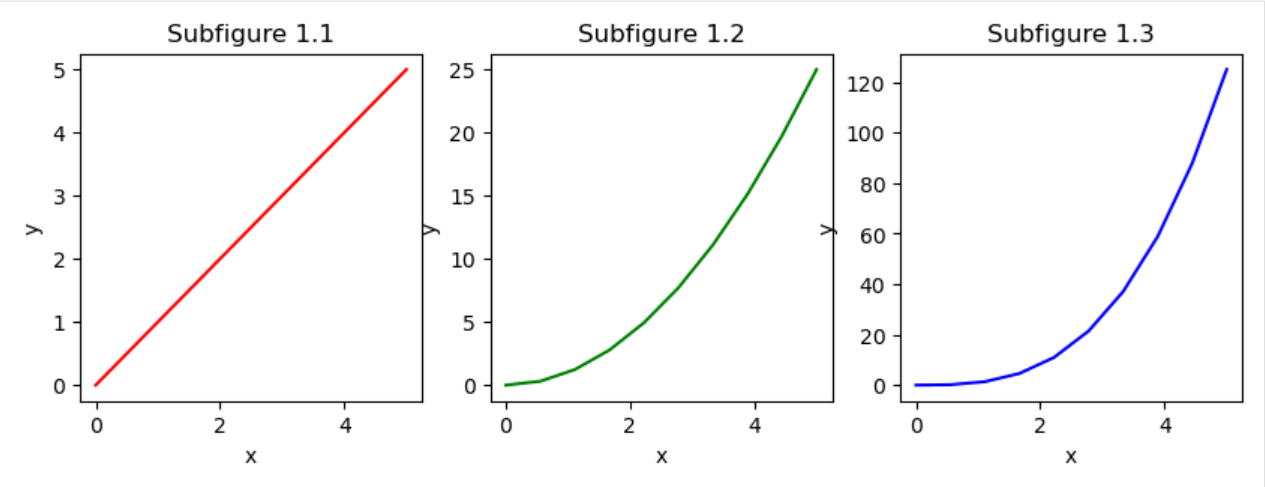
Let's check out the type of the variable `axes` in the above code.

[32]:
```
type(axes)
```

[32]:
```
numpy.ndarray
```

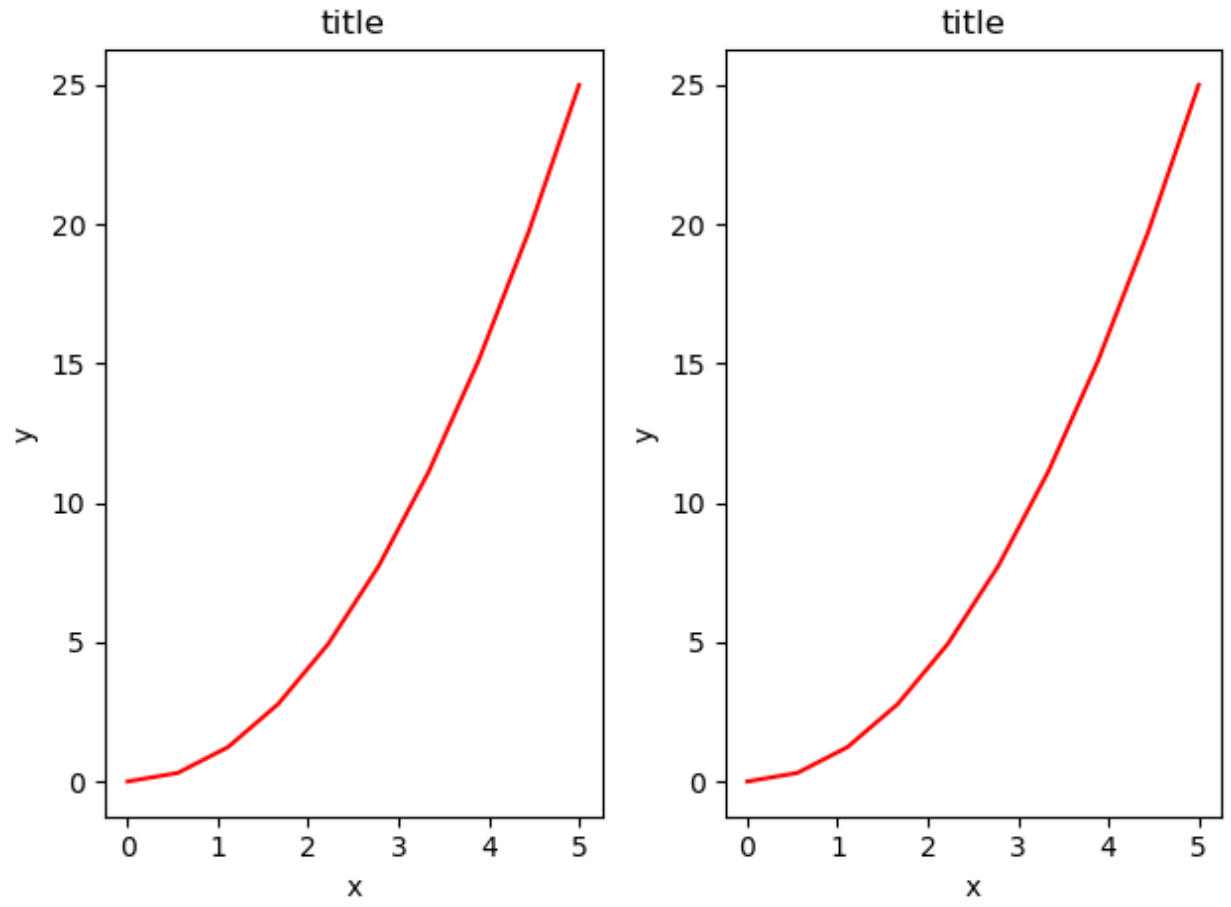
When we create a figure with more than one Axes object, the function `plt.subplots` returns `axes` as a numpy ndarray. To access the individual Axes objects from the numpy array, we can index them `axes[0]`, `axes[1]`, as in the next cell.

[33]:
```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(10, 3))

axes[0].plot(x, x, 'r')
axes[0].set_xlabel('x')
axes[0].set_ylabel('y')
axes[0].set_title('Subfigure 1.1')

axes[1].plot(x, x**2, 'g')
axes[1].set_xlabel('x')
axes[1].set_ylabel('y')
axes[1].set_title('Subfigure 1.2')

axes[2].plot(x, x**3, 'b')
axes[2].set_xlabel('x')
axes[2].set_ylabel('y')
axes[2].set_title('Subfigure 1.3')

plt.show()
```



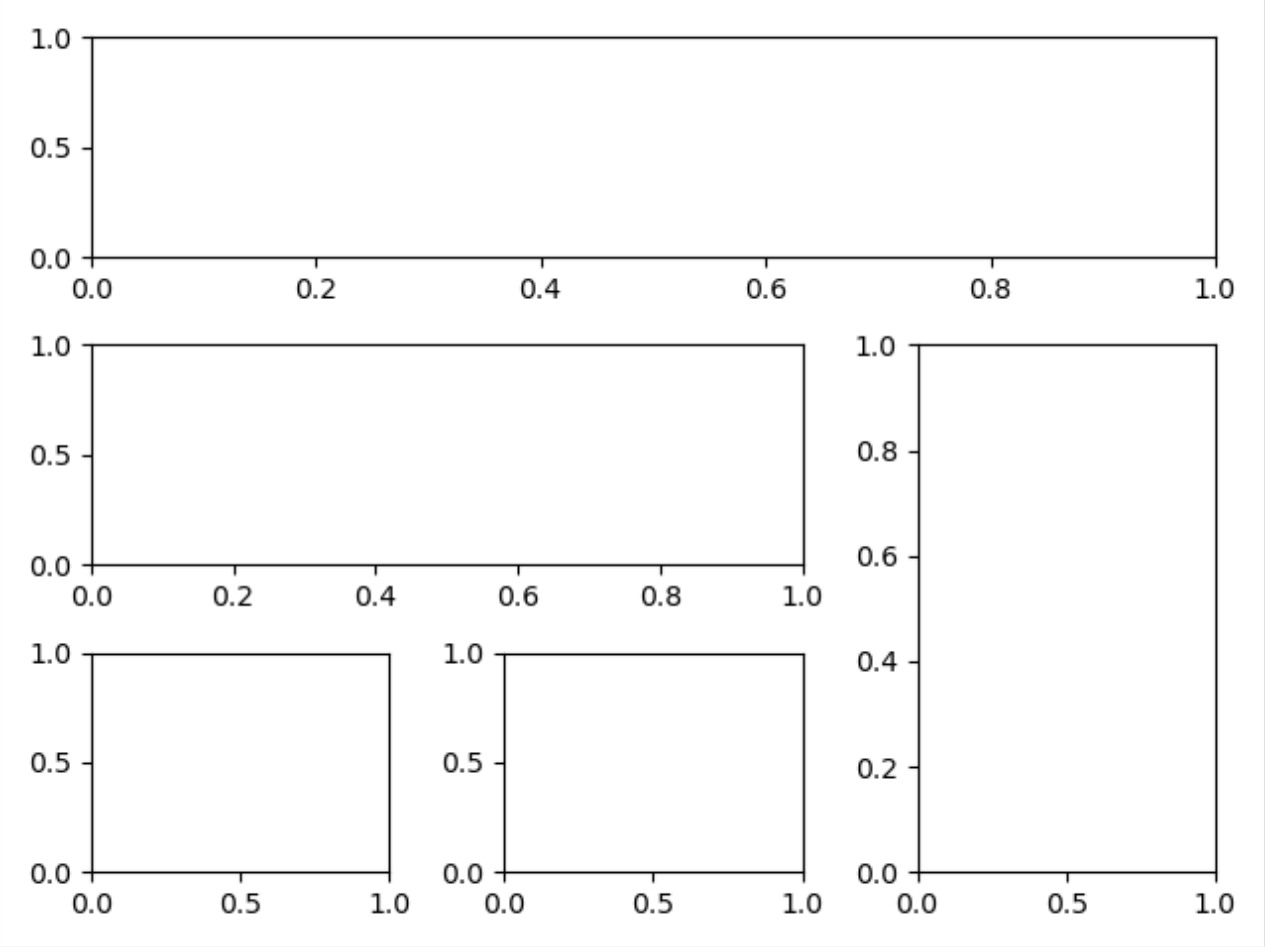
Note also that the above figure has overlapping figures and labels for the y-axis. To deal with that, we can use `fig.tight_layout` method, which automatically adjusts the positions of the axes on the figure canvas so that there is no overlapping content.

[34]:
```
fig, axes = plt.subplots(nrows=1, ncols=2)

for ax in axes:
    ax.plot(x, x**2, 'r')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')

fig.tight_layout()
```



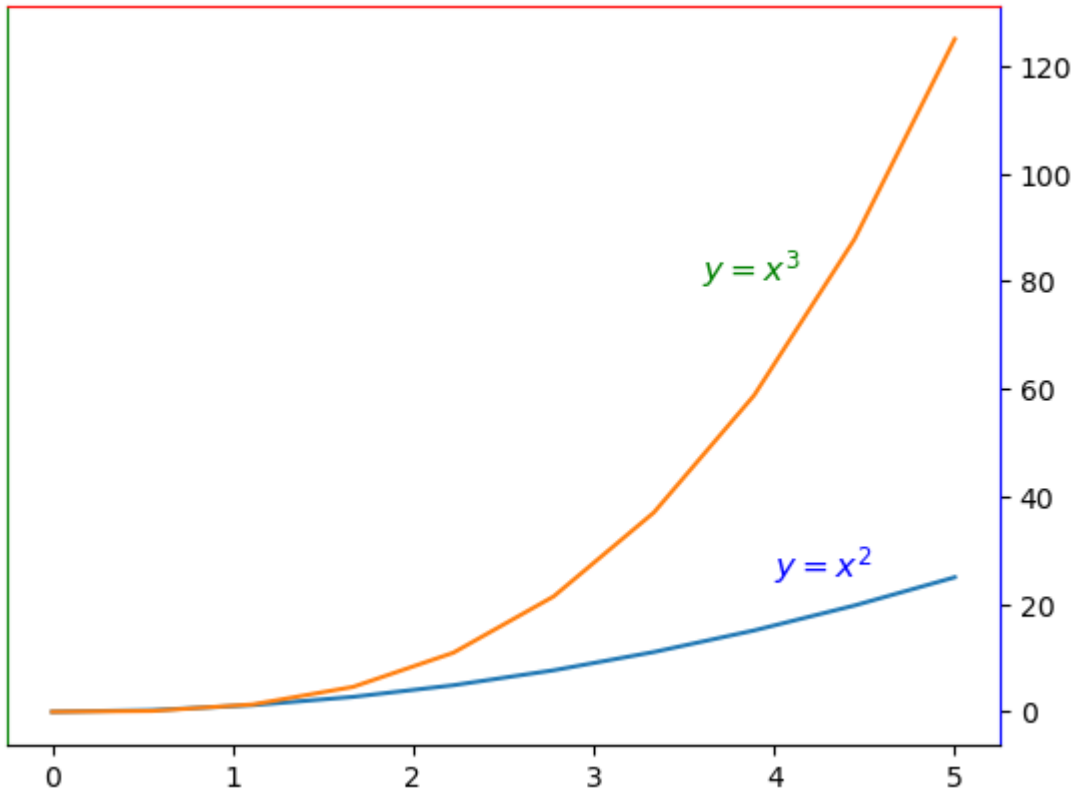
Another way to add sub-figures in Matplotlib is by using the `subplot2grid` function, which allows to specify the length of the span of the subplots across columns or rows.

[35]:
```
fig = plt.figure()
ax1 = plt.subplot2grid((3,3), (0,0), colspan=3)
ax2 = plt.subplot2grid((3,3), (1,0), colspan=2)
ax3 = plt.subplot2grid((3,3), (1,2), rowspan=2)
ax4 = plt.subplot2grid((3,3), (2,0))
ax5 = plt.subplot2grid((3,3), (2,1))
fig.tight_layout()
```



Annotating text in Matplotlib figures can be done using the `text` function. It supports LaTeX formatting just like axis label texts and titles. Also, the `spines` function provides control of the appearance of the borders in the figure.

[36]:
```
fig, ax = plt.subplots()

ax.spines['right'].set_color('blue')
ax.spines['top'].set_color('red')
ax.spines['left'].set_color('green')

ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('right')

ax.plot(x, x**2, x, x**3)

ax.text(4, 25, r"$y=x^2$", fontsize=12, color="blue")
ax.text(3.6, 80, r"$y=x^3$", fontsize=12, color="green")

plt.show()
```



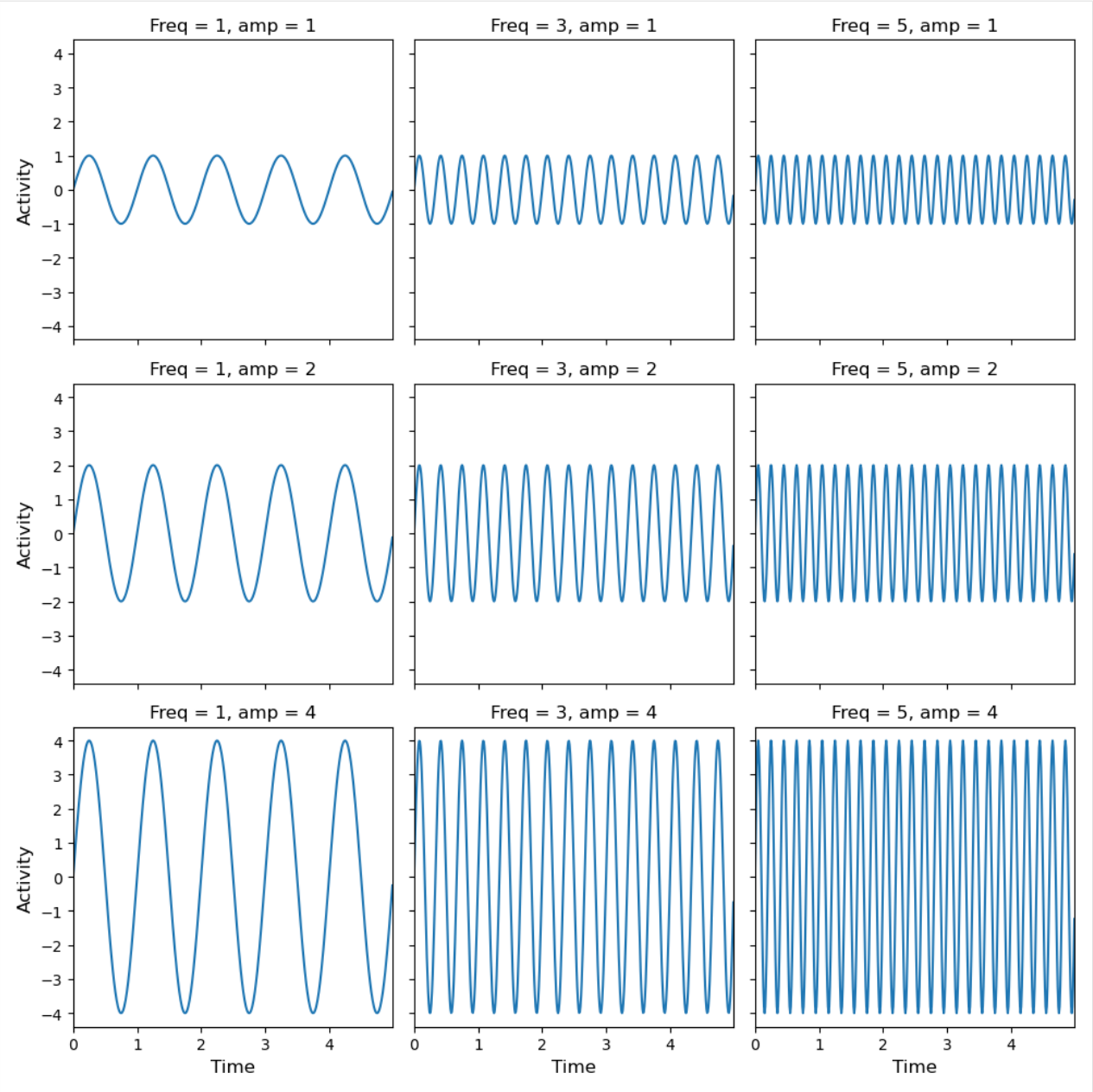
And, here is one more example where sine wave plots are created with different frequencies and amplitudes.

[37]:
```
def create_sine_wave(timepoints, frequency=1, amplitude=1):
    """ Creates a sine wave with a given frequency and amplitude for a given set of␣
↪timepoints.

    Parameters
    ----------
    timepoints : list
        A list with timepoints (assumed to be in seconds)
    frequency : int/float
        Desired frequency (in Hz.)
    amplitude : int/float
        Desired amplitude (arbitrary units)

    Returns
    -------
    sine : list
        A list with floats representing the sine wave
    """
    sine = [amplitude * math.sin(2 * math.pi * frequency * t) for t in timepoints]
    return sine


import math
timepoints = [i / 100 for i in range(500)]
```

(continues on next page)

(continued from previous page)

```
# in the next line, sharex=True and sharey=True are used to force the same range across␣
↪subplots
fig, axes = plt.subplots(ncols=3, nrows=3, figsize=(10, 10), sharex=True, sharey=True)
amps = [1, 2, 4]
freqs = [1, 3, 5]
for i in range(len(amps)):
    for ii in range(len(freqs)):
        sine = create_sine_wave(timepoints, frequency=freqs[ii], amplitude=amps[i])
        axes[i, ii].plot(timepoints, sine)
        axes[i, ii].set_title(f"Freq = {freqs[ii]}, amp = {amps[i]}")
        axes[i, ii].set_xlim(0, max(timepoints))
        if ii == 0:
            axes[i, ii].set_ylabel("Activity", fontsize=12)

        if i == 2:
            axes[i, ii].set_xlabel("Time", fontsize=12)

fig.tight_layout()
plt.show()
```



7.9.8 Appendix

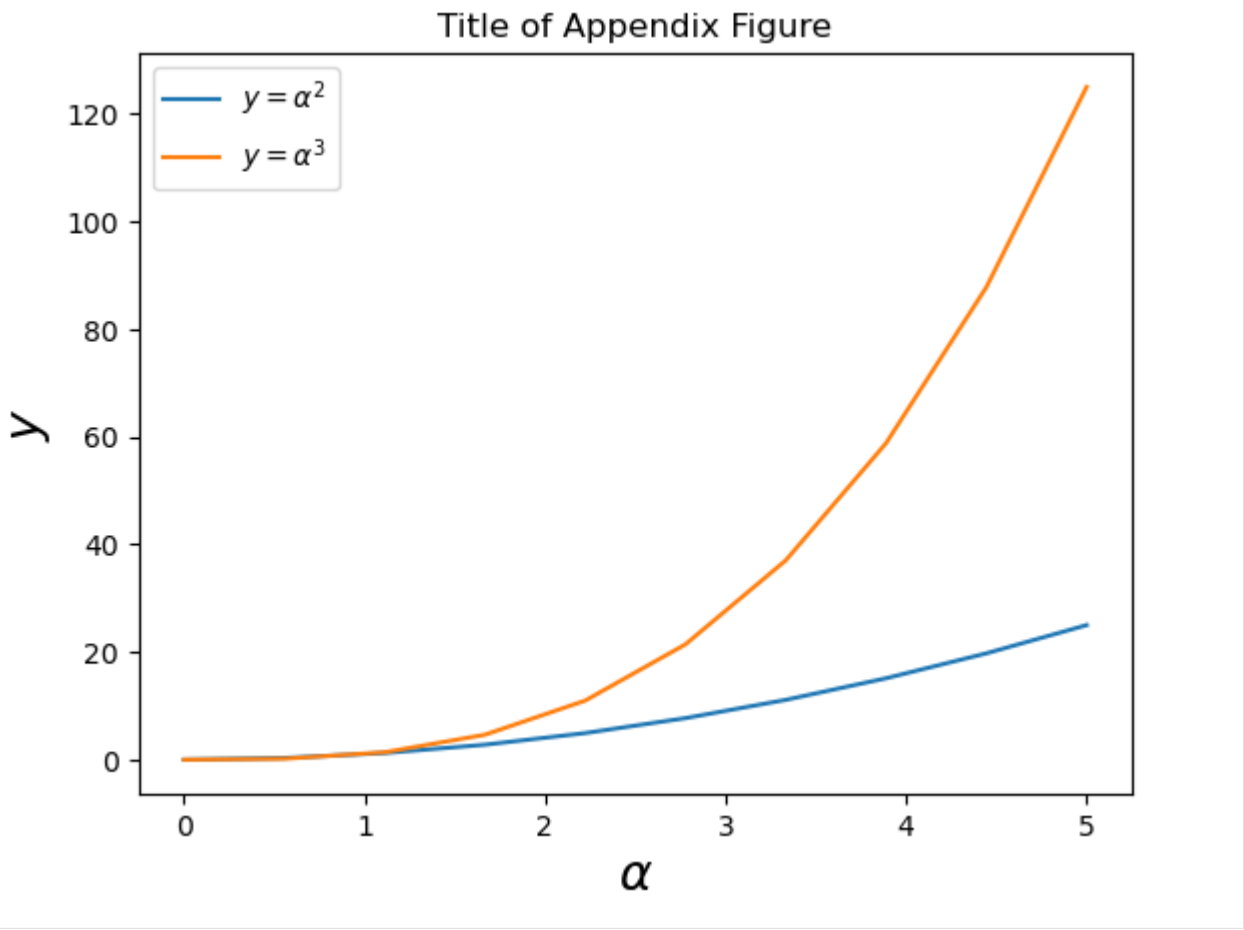
The material in the Appendix is not required for quizzes and assignments.

Formatting Text in Matplotlib with LaTeX

Matplotlib has great support for LaTeX, by using dollar signs to encapsulate LaTeX in any text, such as legend, title, label, etc. (for example, `"$y=x^3$"`).

However, we can run into a slight problem with LaTeX code and Python text strings. In LaTeX, we frequently use the backslash in commands, for example `\alpha` to produce the symbol α. But the backslash already has a meaning in Python strings (the escape code character). To avoid problems in Python with Latex code, we need to use "raw" text strings that are prepended with an 'r', like `r"\alpha"` instead of `"\alpha"`.

[38]:
```
fig, ax = plt.subplots()

ax.plot(x, x**2, label=r"$y = \alpha^2$")
ax.plot(x, x**3, label=r"$y = \alpha^3$")
ax.legend(loc='upper left')
ax.set_xlabel(r'$\alpha$', fontsize=18)
ax.set_ylabel(r'$y$', fontsize=18)
ax.set_title('Title of Appendix Figure');
```



We can also change the global font size and font family, which applies to all text elements in a figure (tick labels, axis labels and titles, legends, etc.).

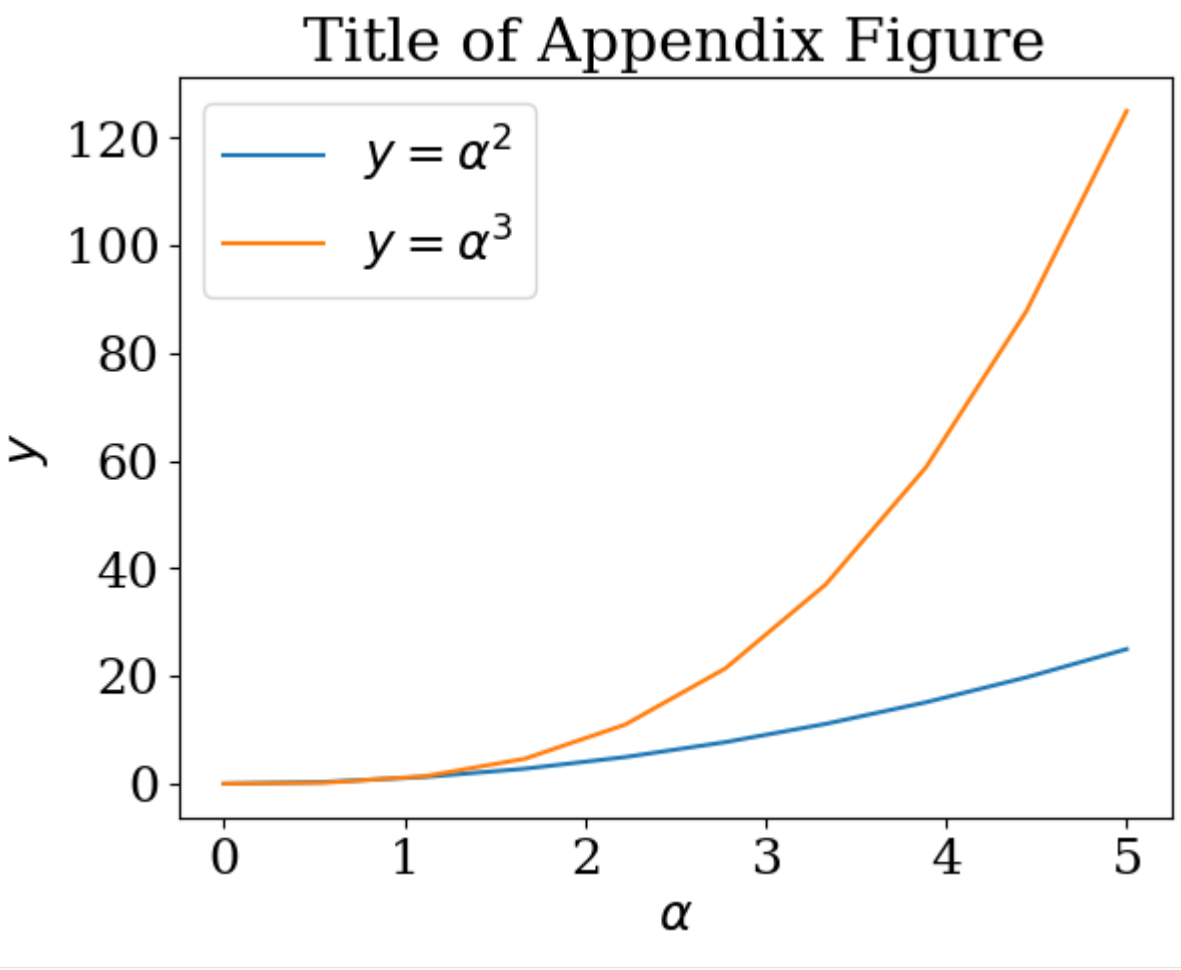
[39]:
```
import matplotlib
```

(continues on next page)

(continued from previous page)

```
# Update the matplotlib configuration parameters:
matplotlib.rcParams.update({'font.size': 18, 'font.family': 'serif'})
```

[40]:
```
fig, ax = plt.subplots()

ax.plot(x, x**2, label=r"$y = \alpha^2$")
ax.plot(x, x**3, label=r"$y = \alpha^3$")
ax.legend(loc=2) # upper left corner
ax.set_xlabel(r'$\alpha$')
ax.set_ylabel(r'$y$')
ax.set_title('Title of Appendix Figure');
```



And, we can restore the font and font size back to the defaults.

[41]:
```
# restore
matplotlib.rcParams.update({'font.size': 12, 'font.family': 'sans'})
```

Colormap and Contour Figures

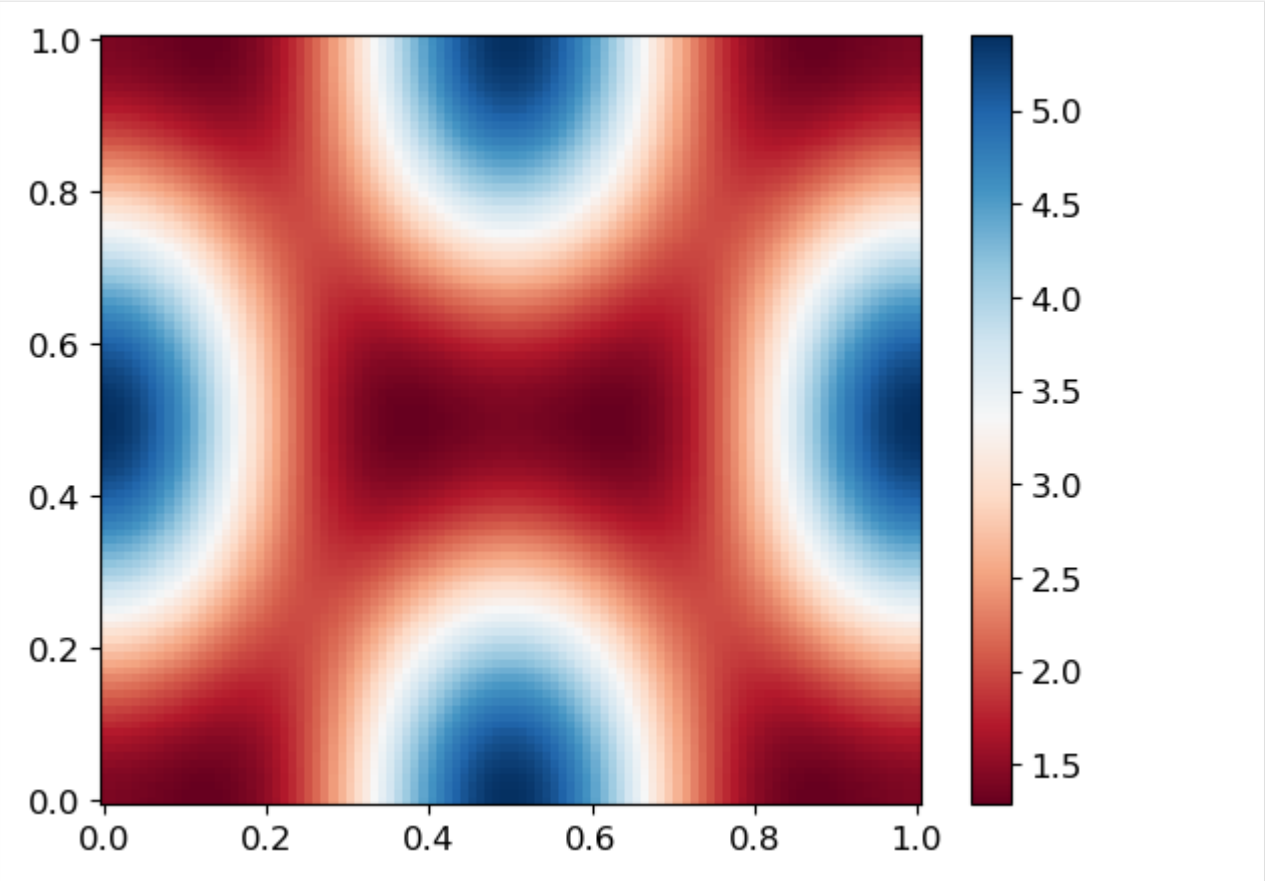
Colormaps and contour figures are useful for plotting functions of two variables. In most of these functions we will use a colormap to encode one dimension of the data. There are a number of predefined colormaps, and it is relatively straightforward to define custom colormaps.

The following cell defines a two-dimensional function, and afterward, plotting the colormap is illustrated with: `pcolor`, `imshow`, and `contour`. The function `fig.colorbar` is used to add a colorbar.

[42]:
```
# Create a two dimensional function
phi_m = np.linspace(0, 2*np.pi, 100)
phi_p = np.linspace(0, 2*np.pi, 100)

alpha = 0.7
phi_ext = 2 * np.pi * 0.5

def flux_qubit_potential(phi_m, phi_p):
    return 2 + alpha - 2 * np.cos(phi_p) * np.cos(phi_m) - alpha * np.cos(phi_ext -␣
↪2*phi_p)

X,Y = np.meshgrid(phi_p, phi_m)
Z = flux_qubit_potential(X, Y).T
```

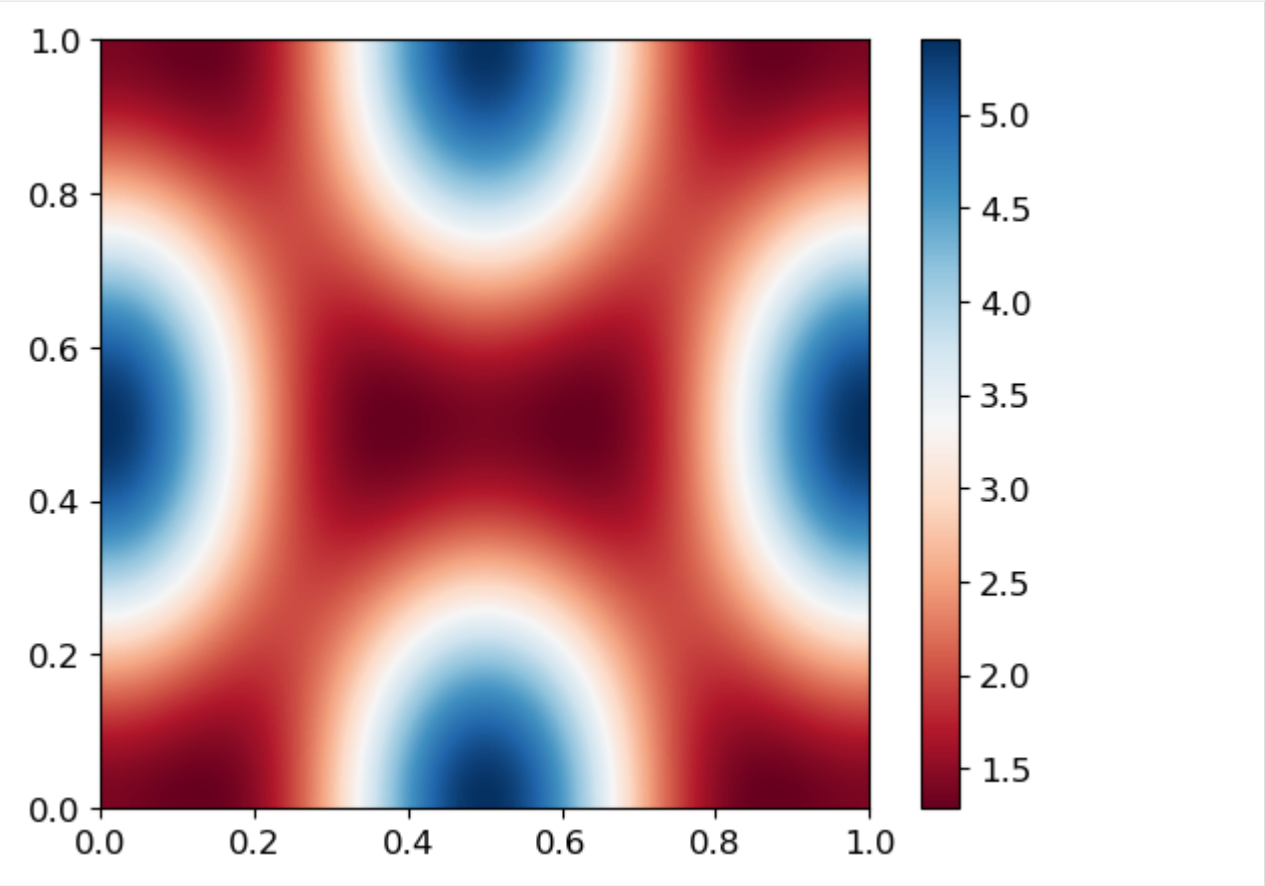
pcolor

[43]:
```
fig = plt.figure()
ax = fig.add_axes([0, 0, 0.8, 0.8])

p = ax.pcolor(X/(2*np.pi), Y/(2*np.pi), Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(),␣
↪vmax=abs(Z).max())
cb = fig.colorbar(p, ax=ax)
```



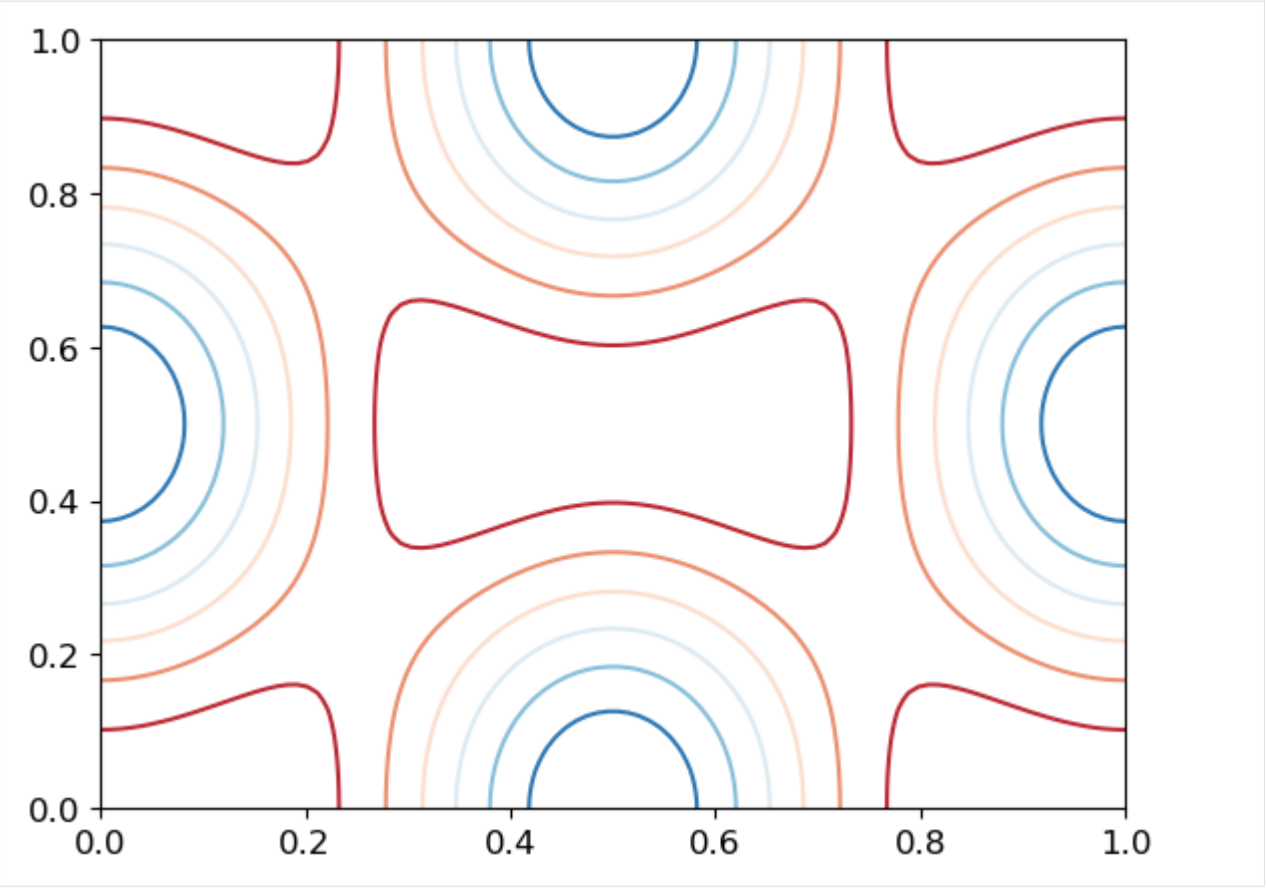
imshow

[44]:
```
fig = plt.figure()
ax = fig.add_axes([0, 0, 0.8, 0.8])

im = ax.imshow(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).max(),␣
↪extent=[0, 1, 0, 1])
im.set_interpolation('bilinear')

cb = fig.colorbar(im, ax=ax)
```



contour

[45]:
```
fig = plt.figure()
ax = fig.add_axes([0, 0, 0.8, 0.8])

cnt = ax.contour(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).max(),␣
↪extent=[0, 1, 0, 1])
```



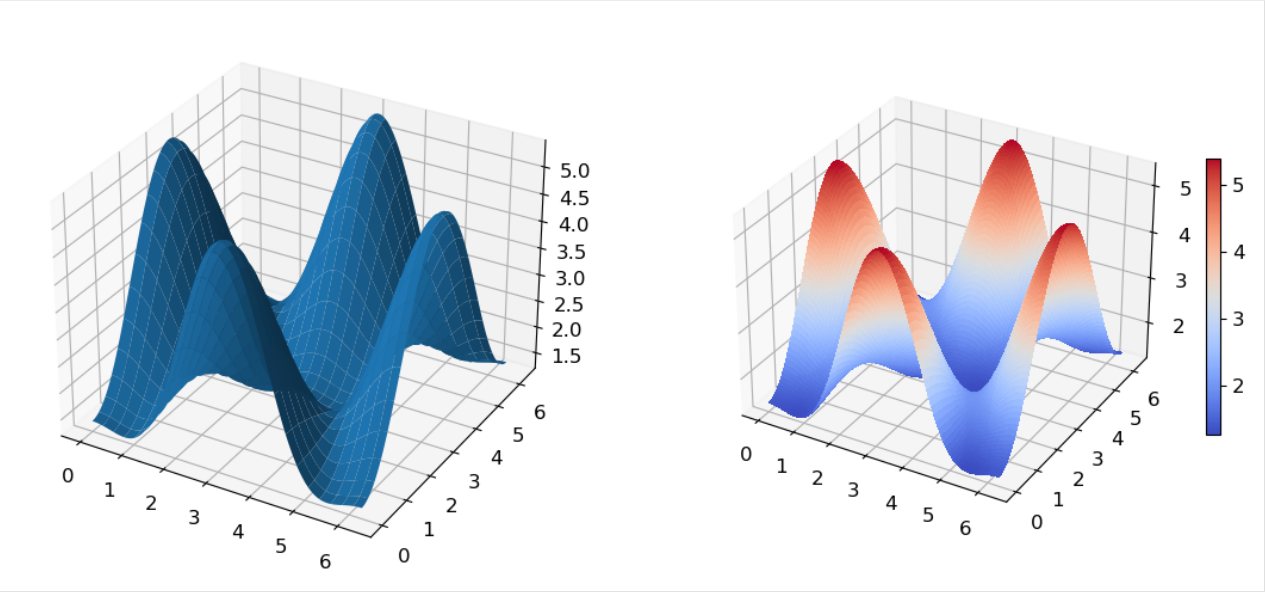
3D Figures

To use 3D graphics in Matplotlib, we first need to create an instance of the `Axes3D` class. 3D axes can be added to a matplotlib figure in exactly the same way as 2D axes. Or, they can be added by passing a `projection='3d'` keyword argument to the `add_axes` or `add_subplot` methods.

[46]:
```
from mpl_toolkits.mplot3d.axes3d import Axes3D
```

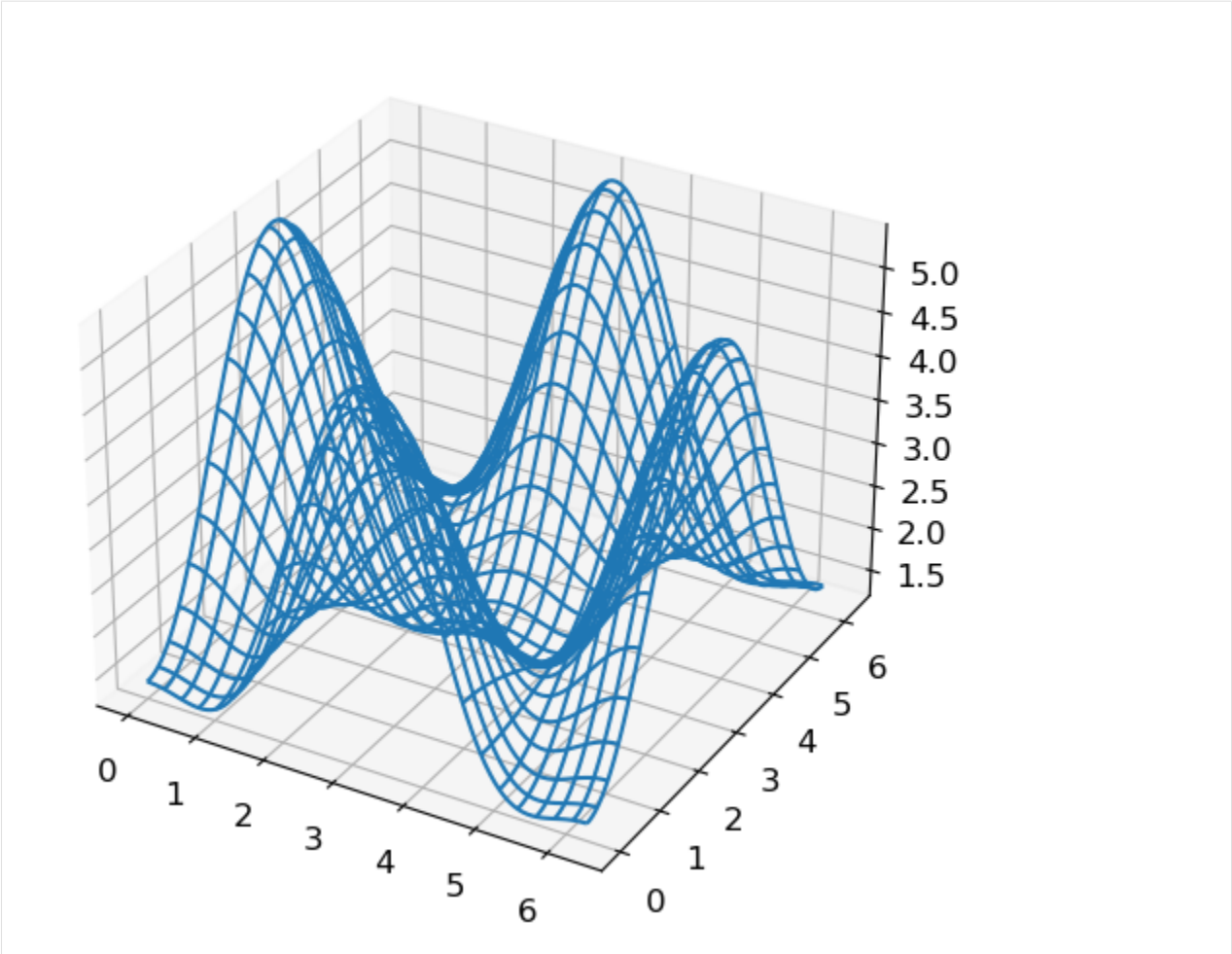
Surface plots

[47]:
```
fig = plt.figure(figsize=(14,6))

# `ax` is a 3D-aware axis instance because of the projection='3d' keyword argument to add_
↪subplot
ax = fig.add_subplot(1, 2, 1, projection='3d')
p = ax.plot_surface(X, Y, Z, rstride=4, cstride=4, linewidth=0)

# surface_plot with color grading and color bar
ax = fig.add_subplot(1, 2, 2, projection='3d')
p = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=matplotlib.cm.coolwarm,␣
↪linewidth=0, antialiased=False)
cb = fig.colorbar(p, shrink=0.5)
```



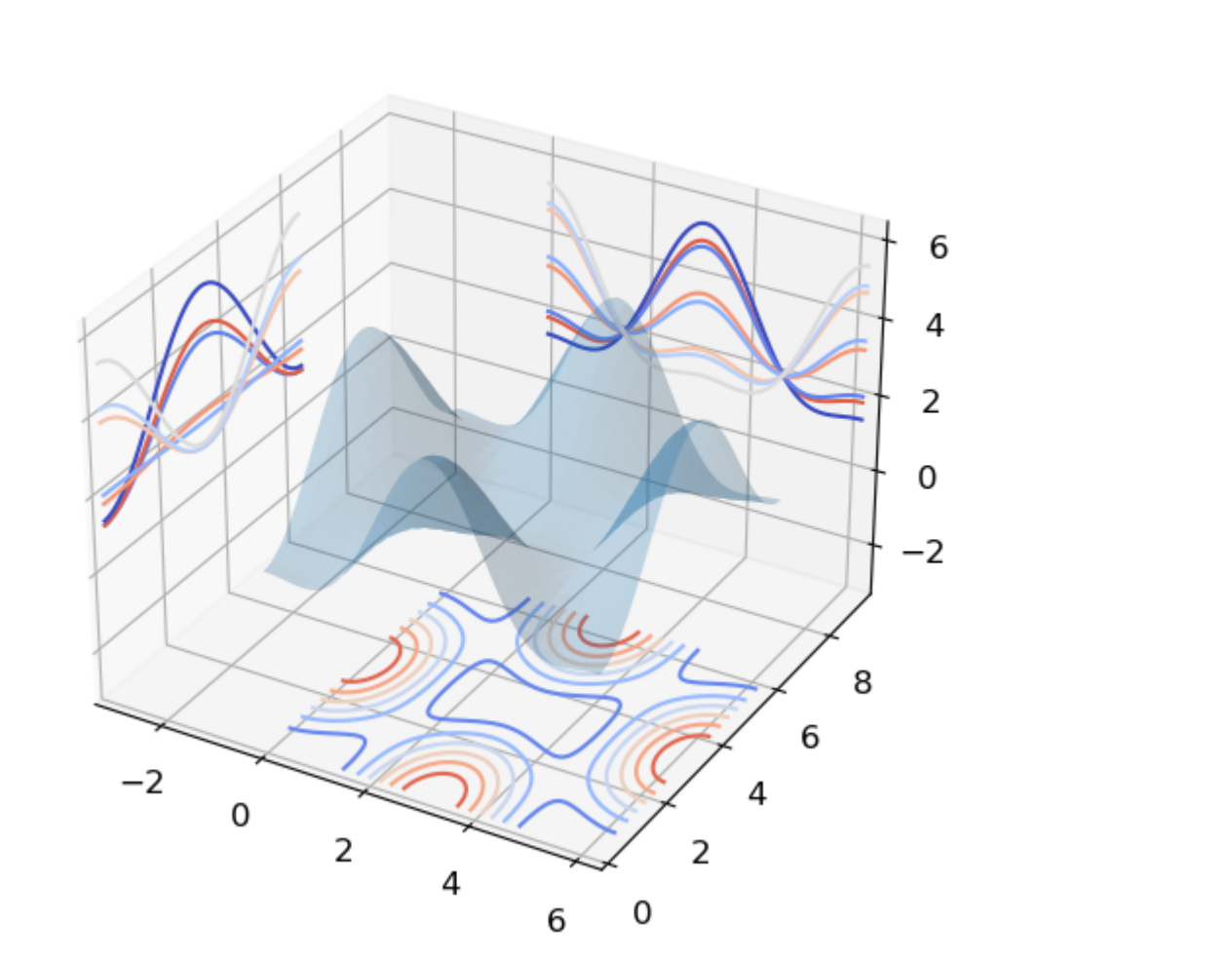
Wire-frame plot

[48]:
```
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1, 1, 1, projection='3d')

p = ax.plot_wireframe(X, Y, Z, rstride=4, cstride=4)
```



Contour plots with projections

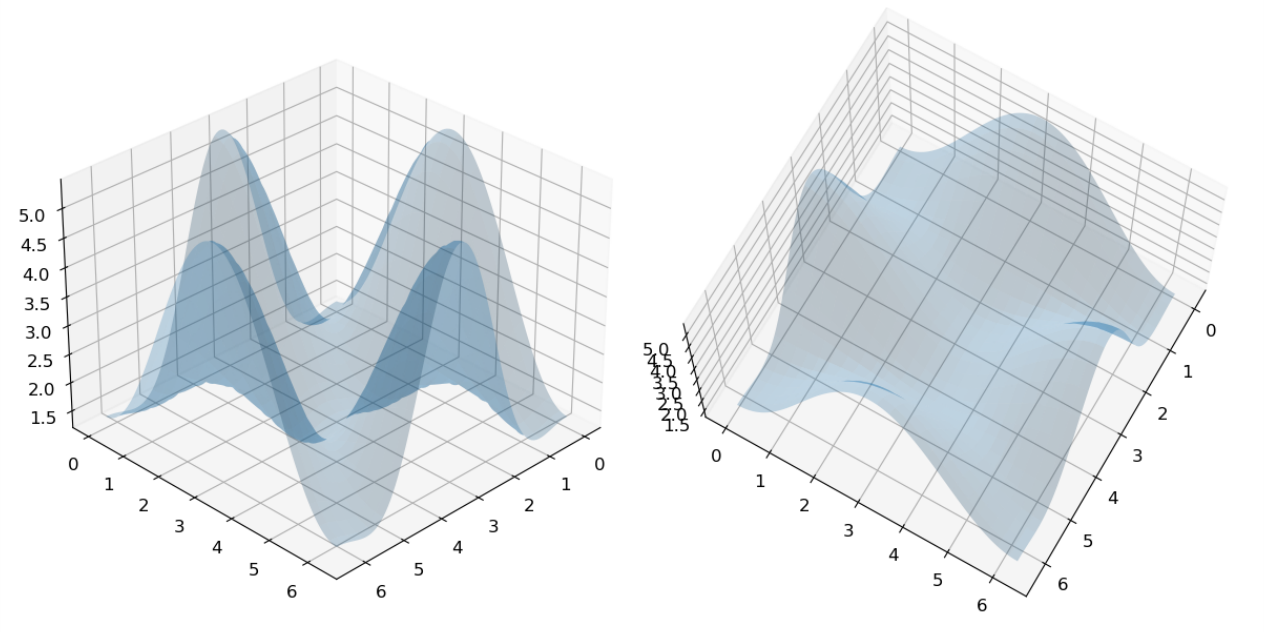
[49]:
```
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1,1,1, projection='3d')

ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
cset = ax.contour(X, Y, Z, zdir='z', offset=-np.pi, cmap=matplotlib.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-np.pi, cmap=matplotlib.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=3*np.pi, cmap=matplotlib.cm.coolwarm)

ax.set_xlim3d(-np.pi, 2*np.pi);
ax.set_ylim3d(0, 3*np.pi);
ax.set_zlim3d(-np.pi, 2*np.pi);
```



Change the view angle

We can change the perspective of a 3D plot using the `view_init` method, which takes two arguments: `elevation` and `azimuth` angle (in degrees).

[50]:
```
fig = plt.figure(figsize=(12,6))

ax = fig.add_subplot(1,2,1, projection='3d')
ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
ax.view_init(30, 45)

ax = fig.add_subplot(1,2,2, projection='3d')
ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
ax.view_init(70, 30)

fig.tight_layout()
```



7.9.9 References

1. Matplotlib - Pyplot Tutorial, available at: https://matplotlib.org/2.0.2/users/pyplot_tutorial.html.
2. Lectures on Scientific Computing with Python, Robert Johansson, available at: https://github.com/jrjohansson/scientific-python-lectures/blob/master/Lecture-4-Matplotlib.ipynb.
3. Introduction to Matplotlib (tutorial), by Lukas Snoek, available at: https://lukas-snoek.com/introPy/solutions/week_1/2_matplotlib.html.
4. Matplotlib - An Intro to Creating Graphs with Python, Mike Driscol, available at: https://www.blog.pythonlibrary.org/2021/09/07/matplotlib-an-intro-to-creating-graphs-with-python/.
5. Scientific Computing in Python: Introduction to NumPy and Matplotlib, by Sebastian Raschka, available at: https://sebastianraschka.com/blog/2020/numpy-intro.html.
6. Python Tutorial, Visualization with Matplotilb, available at: https://github.com/zhiyzuo/python-tutorial/blob/master/4-Visualization-with-Matplotlib.ipynb.
7. Numpy, Pandas, Matplotlib Tutorial, available at https://github.com/veb-101/Numpy-Pandas-Matplotlib-Tutorial.

BACK TO TOP

7.10 Lecture 10 - Databases and SQL

7.10.1 10.1 Introduction to SQL

SQL (Structured Query Language) is a programming language designed for managing data in Relational Data Base Management Systems (RDBMS), or for stream processing in Relational Data Stream Management Systems (RDSMS). A *relational database* is a database that stores related information across multiple tables, and allows to query information in more than one table at the same time. Within a table, the data is organized in a tabular format with rows and columns.

SQL was initially developed in 1970, and since then different companies and vendors implemented SQL in their products with some variations. To bring greater conformity in the variants of SQL, the American National Standards Institute (ANSI) published the first SQL standard in 1986. The standard has been updated every few years since then.

Today, there are several variants of SQL for database management systems available, some of which were developed by companies such as IBM and Oracle, as well as there are variants developed by communities, such as MySQL, PostgreSQL, MariaDB, etc. Although these variants of SQL have certain differences, they are based on the basic SQL syntax, and are quite similar.

The main advantages of SQL include standardized syntax (since all relational database systems have an SQL query interpreter built-in), and is easy-to-understand due to using English-like commands and functions.

Relational Databases

Relational databases store information in multiple tables, which allows to work with more complex data, and have flexibility in the way the data is organized. An example is shown in the next figure, where a database is shown that is used for managing the HR data of a small business.

This database has seven tables:

- Jobs table stores data related to job title and salary range.
- Employees table stores the data of employees.
- Dependents table stores the employee's dependents.
- Departments table stores department data.
- Regions table stores the data of regions such as Asia, Europe, America, Middle East, and Africa.

- Countries table stores the data of countries where the company is doing business.
- Locations table stores the location of the departments of the company.

Each table contains many records with rows and columns (similar to an Excel spreadsheet), and the records have relationships across the tables. Using multiple tables in a relational database allows us to avoid duplication of information, in comparison to using a single table to store all information. Also, it provides flexibility in how we work with the data. To establish relationships between the records in different tables we need to use an ID or identifier for each employee. The identifier for each employee, or in general for each record (row) in a relational database, is referred to as *primary key*. For instance, each employee can be assigned an ID value (such as employee 162), and each table would have an ID column (primary key column) to establish the relationship with the other tables in the database.

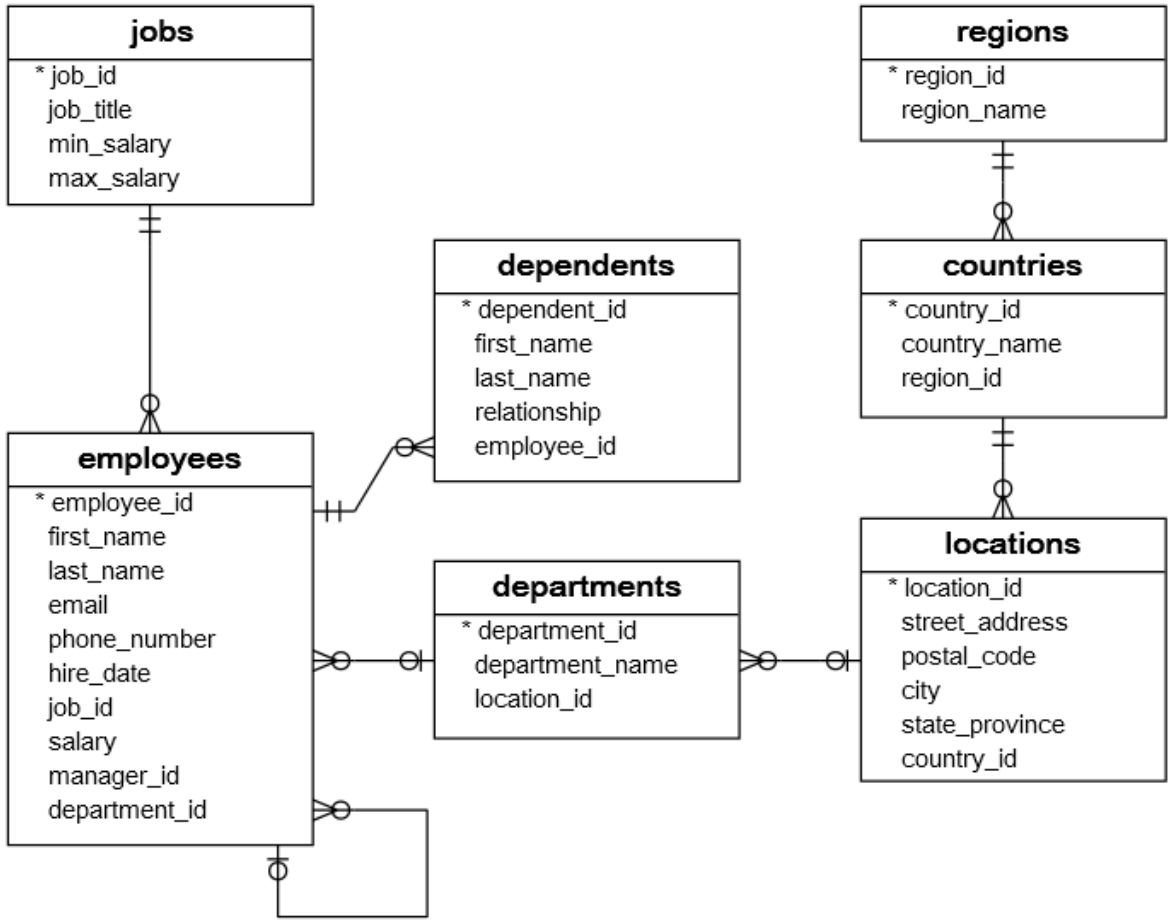


Figure source: Reference [1].

SQL for Data Science

SQL is a very important tool for data scientists, data analysts, developers, and database administrators. In particular, as many companies become data-driven, SQL becomes an essential tool for handling data stored in databases and performing various data analytics operations, such as calculating data statistics, updating records, removing duplicate columns, calculating correlations between records, and similar.

SQL versus Pandas

SQL has similarities with the Pandas library, as it offers similar functionality to Pandas, which includes data manipulation over rows and columns, data merging, grouping, dealing with missing values, and similar. However, Pandas is not a relational database management library, but it is a data frame library.

Still, Pandas offers additional functions and flexibility for handling and manipulating tabular data, and many users download databases to their local machine, and afterward use Pandas for data processing, rather than using SQL to process the data on the server.

The benefits of using SQL over Pandas can depend on the task. Several considerations include:

- In the case of a large database of information (e.g., GigaBytes of data), downloading the database to the local machine to be processed by Pandas may be slow or infeasible. Pandas is more suitable for processing small to medium size databases in Python.
- Even if the user can download the data on the local machine for processing with Pandas, it may be required to apply some level of preprocessing or organizing the data on the server using SQL.
- Some tasks can require that the data processing is done in the existing database. Also, when the tasks require fast data retrieval and processing, SQL can be more efficient than Pandas.

SQLite

In this lecture, we will use **SQLite** which implements a self-contained, serverless SQL database engine. SQLite is lightweight in terms of setup and required recourses. Unlike most other SQL variants, SQLite does not have a separate server process, and it reads and writes directly to disk files. That is, it does not use the client/server model. Because it has no server managing access to it, SQLite is not suitable in multiuser environments where multiple people can simultaneously edit files.

7.10.2 10.2 Using SQLite with Python

To demonstrate the use of SQLite with Python, in this lecture we will use *magic commands* in Jupyter Notebook. Magic commands are special commands which are not valid Python code, but perform certain actions in a Jupyter Notebook. They begin with the `%` symbol.

The library `ipython-sql` offers the magic functions `%sql` and `%%sql`, which allow to connect to a database and use standard SQL commands in Jupyter Notebooks. To run `ipython-sql` on your computer, it needs to be installed (e.g., by `pip install ipython-sql`). If we run the notebooks on Google Colab, `ipython-sql` comes preinstalled.

To load the `ipython-sql` library we will use `%load_ext sql` as in the next code. `%load_ext` is a magic command that loads an external package that can add new magic commands.

[1]:
```
%load_ext sql
```

The magic command `%sql` is used to execute an SQL query that is contained in a single statement in a Jupyter notebook cell, and `%%sql` allows to execute an SQL query that is contained in multiple SQL statements in a single cell.

7.10.3 10.3 Create a New Table

To create a new table we will use the SQL command `CREATE TABLE`, as shown in the next cell. If the table already exists in the database, an error message will show up.

In order to establish a connection to the newly created table, we used `%%sql sqlite://` in the cell below. If we wanted to create a new table in an existing database to which we have already established a connection, we could have used only the magic command `%%sql`.

SQL has many commands or keywords that have special meaning, such as SELECT, INSERT, DELETE, and these keywords cannot be used as names of tables, columns, or other objects.

To make SQL language more readable, it is a convention to write the SQL commands with uppercase letters, and the other variables and identifiers with lowercase letters. However, this is not required, as the SQL commands are not case-sensitive.

Let's create a table called `cars` which has 3 columns: `id`, `name`, and `price`. In the cell below, we specify that the values of `id` and `price` columns are integers, denoted by the `INTEGER` keyword. The names in the `name` column have a character string type (`VARCHAR`, i.e., variable-length character string) and should have at most 50 characters. We also specified with `NOT NULL` that the values should not be missing in the `id` and `name` columns, i.e., when we insert data into the table we have to specify the values for the `NON NULL` columns.

Each table can have only one **PRIMARY KEY** column that uniquely identifies each row in the table, and prevents from inserting duplicate rows in the table. For the table below we set *PRIMARY KEY* to the `id` column. It is not required to define a primary key, however, it is a good practice to do it for every table.

[2]:
```
%%sql sqlite://
CREATE TABLE cars(
    id INTEGER NOT NULL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    price INTEGER);
```

```
Done.
```

[2]:
```
[]
```

The table that we just created is empty, and to add data to the table we will use the `INSERT` statement. In each row we provide values for `id`, `name`, and `price`.

When using multiple statements in SQL, the statements in each line need to be separated with a semicolon `;`. The last statement in a cell does not have to be followed by a semicolon.

Inline comments can be inserted by using two consecutive hyphens `--` that comment the rest of the line, as shown in the second line of the next cell.

And also, comments that span multiple lines can be inserted by using the multiline C-style notation `/* comment */` as in the last lines in the cell.

Note also that we used just the magic command `%%sql` in this cell, and we didn't need to write `%%sql sqlite://` as in the above cell. The reason is that in the above cell we used `%%sql sqlite://` to establish a connection to the newly created table. Once a connection is established, we can use only `%sql` or `%%sql` to work with the table.

[3]:
```
%%sql
INSERT INTO cars VALUES(1,'Audi',52642); --Two consecutive hyphens (--) comment the rest␣
↪of the line
INSERT INTO cars VALUES(2,'Mercedes',57127);
INSERT INTO cars VALUES(3,'Skoda',9000);
INSERT INTO cars VALUES(4,'Volvo',29000);
INSERT INTO cars VALUES(5,'Bentley',350000);
```

(continues on next page)

(continued from previous page)

```
INSERT INTO cars VALUES(6,'Citroen',21000);
INSERT INTO cars VALUES(7,'Hummer',41400);
INSERT INTO cars VALUES(8,'Volkswagen',21600);
/* A comment that spans
more than one line */
```

```
 * sqlite://
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
Done.
```

[3]:
```
[]
```

We can display the table with the following code. Notice again that we used a single % in the magic command `%sql`, since we have only one line of code.

[4]:
```
%sql SELECT * from cars
```

```
 * sqlite://
Done.
```

[4]:
```
[(1, 'Audi', 52642),
 (2, 'Mercedes', 57127),
 (3, 'Skoda', 9000),
 (4, 'Volvo', 29000),
 (5, 'Bentley', 350000),
 (6, 'Citroen', 21000),
 (7, 'Hummer', 41400),
 (8, 'Volkswagen', 21600)]
```

Another Example of Creating a Table

In the next simple example, we will create another table called `writer`, with columns `FirstName`, `LastName`, and `Year`.

[5]:
```
%%sql sqlite://
CREATE TABLE writer(
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    Year INTEGER NOT NULL PRIMARY KEY);
```

```
Done.
```

[5]:
```
[]
```

[6]:
```
%%sql
INSERT INTO writer VALUES ('William', 'Shakespeare', 1616);
```

(continues on next page)

(continued from previous page)

```
INSERT INTO writer VALUES ('Lin', 'Han', 1996);
INSERT INTO writer VALUES ('Peter', 'Brecht', 1978);
```

```
 * sqlite://
1 rows affected.
1 rows affected.
1 rows affected.
```

[6]:
```
[]
```

[7]:
```
%sql SELECT * from writer
```

```
 * sqlite://
Done.
```

[7]:
```
[('William', 'Shakespeare', 1616),
 ('Peter', 'Brecht', 1978),
 ('Lin', 'Han', 1996)]
```

7.10.4 10.4 Database Example

As an example of a database, let's create a database that was shown in the above section, related to managing the HR data of a small business.

The cells below first create the tables (recall that the database has 7 tables), and afterward the information for each table is inserted.

[8]:
```
%%sql sqlite://

CREATE TABLE regions (
    region_id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    region_name TEXT NOT NULL);

CREATE TABLE countries (
    country_id TEXT NOT NULL,
    country_name TEXT NOT NULL,
    region_id INTEGER NOT NULL,
    PRIMARY KEY (country_id ASC),
    FOREIGN KEY (region_id) REFERENCES regions (region_id) ON DELETE CASCADE ON UPDATE␣
↪CASCADE);

CREATE TABLE locations (
    location_id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    street_address TEXT,
    postal_code TEXT,
    city text NOT NULL,
    state_province TEXT,
    country_id INTEGER NOT NULL,
    FOREIGN KEY (country_id) REFERENCES countries (country_id) ON DELETE CASCADE ON␣
↪UPDATE CASCADE);

CREATE TABLE departments (
    department_id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
```

(continues on next page)

(continued from previous page)

```
    department_name TEXT NOT NULL,
    location_id INTEGER NOT NULL,
    FOREIGN KEY (location_id) REFERENCES locations (location_id) ON DELETE CASCADE ON␣
↪UPDATE CASCADE);

CREATE TABLE jobs (
    job_id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    job_title TEXT NOT NULL,
    min_salary DOUBLE NOT NULL,
    max_salary DOUBLE NOT NULL);

CREATE TABLE employees (
    employee_id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    first_name TEXT,
    last_name TEXT NOT NULL,
    email TEXT NOT NULL,
    phone_number TEXT,
    hire_date TEXT NOT NULL,
    job_id INTEGER NOT NULL,
    salary DOUBLE NOT NULL,
    manager_id INTEGER,
    department_id INTEGER NOT NULL,
    FOREIGN KEY (job_id) REFERENCES jobs (job_id) ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (department_id) REFERENCES departments (department_id) ON DELETE CASCADE␣
↪ON UPDATE CASCADE,
    FOREIGN KEY (manager_id) REFERENCES employees (employee_id) ON DELETE CASCADE ON␣
↪UPDATE CASCADE);

CREATE TABLE dependents (
    dependent_id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    relationship TEXT NOT NULL,
    employee_id INTEGER NOT NULL,
    FOREIGN KEY (employee_id) REFERENCES employees (employee_id) ON DELETE CASCADE ON␣
↪UPDATE CASCADE);
```

```
Done.
Done.
Done.
Done.
Done.
Done.
Done.
```

[8]:
```
[]
```

[ ]:
```
%%sql

/*Data for the table regions */
INSERT INTO regions(region_id,region_name) VALUES (1,'Europe');
INSERT INTO regions(region_id,region_name) VALUES (2,'Americas');
INSERT INTO regions(region_id,region_name) VALUES (3,'Asia');
```

(continues on next page)

(continued from previous page)

```
INSERT INTO regions(region_id,region_name) VALUES (4,'Middle East and Africa');

/*Data for the table countries */
INSERT INTO countries(country_id,country_name,region_id) VALUES ('AR','Argentina',2);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('AU','Australia',3);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('BE','Belgium',1);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('BR','Brazil',2);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('CA','Canada',2);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('CH','Switzerland',1);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('CN','China',3);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('DE','Germany',1);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('DK','Denmark',1);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('EG','Egypt',4);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('FR','France',1);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('HK','HongKong',3);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('IL','Israel',4);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('IN','India',3);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('IT','Italy',1);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('JP','Japan',3);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('KW','Kuwait',4);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('MX','Mexico',2);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('NG','Nigeria',4);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('NL','Netherlands',1);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('SG','Singapore',3);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('UK','United Kingdom',
↪1);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('US','United States of␣
↪America',2);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('ZM','Zambia',4);
INSERT INTO countries(country_id,country_name,region_id) VALUES ('ZW','Zimbabwe',4);

/*Data for the table locations */
INSERT INTO locations(location_id,street_address,postal_code,city,state_province,country_
↪id) VALUES (1400,'2014 Jabberwocky Rd','26192','Southlake','Texas','US');
INSERT INTO locations(location_id,street_address,postal_code,city,state_province,country_
↪id) VALUES (1500,'2011 Interiors Blvd','99236','South San Francisco','California','US
↪');
INSERT INTO locations(location_id,street_address,postal_code,city,state_province,country_
↪id) VALUES (1700,'2004 Charade Rd','98199','Seattle','Washington','US');
INSERT INTO locations(location_id,street_address,postal_code,city,state_province,country_
↪id) VALUES (1800,'147 Spadina Ave','M5V 2L7','Toronto','Ontario','CA');
INSERT INTO locations(location_id,street_address,postal_code,city,state_province,country_
↪id) VALUES (2400,'8204 Arthur St',NULL,'London',NULL,'UK');
INSERT INTO locations(location_id,street_address,postal_code,city,state_province,country_
↪id) VALUES (2500,'Magdalen Centre, The Oxford Science Park','OX9 9ZB','Oxford','Oxford
↪','UK');
INSERT INTO locations(location_id,street_address,postal_code,city,state_province,country_
↪id) VALUES (2700,'Schwanthalerstr. 7031','80925','Munich','Bavaria','DE');

/*Data for the table jobs */
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (1,'Public Accountant',
↪4200.00,9000.00);
```

(continues on next page)

(continued from previous page)

```
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (2,'Accounting Manager',
↪8200.00,16000.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (3,'Administration␣
↪Assistant',3000.00,6000.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (4,'President',20000.00,
↪40000.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (5,'Administration Vice␣
↪President',15000.00,30000.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (6,'Accountant',4200.00,
↪9000.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (7,'Finance Manager',
↪8200.00,16000.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (8,'Human Resources␣
↪Representative',4000.00,9000.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (9,'Programmer',4000.00,
↪10000.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (10,'Marketing Manager',
↪9000.00,15000.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (11,'Marketing␣
↪Representative',4000.00,9000.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (12,'Public Relations␣
↪Representative',4500.00,10500.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (13,'Purchasing Clerk',
↪2500.00,5500.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (14,'Purchasing Manager',
↪8000.00,15000.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (15,'Sales Manager',
↪10000.00,20000.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (16,'Sales Representative
↪',6000.00,12000.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (17,'Shipping Clerk',
↪2500.00,5500.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (18,'Stock Clerk',2000.
↪00,5000.00);
INSERT INTO jobs(job_id,job_title,min_salary,max_salary) VALUES (19,'Stock Manager',5500.
↪00,8500.00);

/*Data for the table departments */
INSERT INTO departments(department_id,department_name,location_id) VALUES (1,
↪'Administration',1700);
INSERT INTO departments(department_id,department_name,location_id) VALUES (2,'Marketing',
↪1800);
INSERT INTO departments(department_id,department_name,location_id) VALUES (3,'Purchasing
↪',1700);
INSERT INTO departments(department_id,department_name,location_id) VALUES (4,'Human␣
↪Resources',2400);
INSERT INTO departments(department_id,department_name,location_id) VALUES (5,'Shipping',
↪1500);
INSERT INTO departments(department_id,department_name,location_id) VALUES (6,'IT',1400);
INSERT INTO departments(department_id,department_name,location_id) VALUES (7,'Public␣
↪Relations',2700);
INSERT INTO departments(department_id,department_name,location_id) VALUES (8,'Sales',
↪2500);
```

(continues on next page)

(continued from previous page)

```
INSERT INTO departments(department_id,department_name,location_id) VALUES (9,'Executive',
↪1700);
INSERT INTO departments(department_id,department_name,location_id) VALUES (10,'Finance',
↪1700);
INSERT INTO departments(department_id,department_name,location_id) VALUES (11,'Accounting
↪',1700);

/*Data for the table employees */
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (100,'Steven','King','steven.
↪king@sqltutorial.org','515.123.4567','1987-06-17',4,24000.00,NULL,9);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (101,'Neena','Kochhar','neena.
↪kochhar@sqltutorial.org','515.123.4568','1989-09-21',5,17000.00,100,9);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (102,'Lex','De Haan','lex.de␣
↪haan@sqltutorial.org','515.123.4569','1993-01-13',5,17000.00,100,9);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (103,'Alexander','Hunold','alexander.
↪hunold@sqltutorial.org','590.423.4567','1990-01-03',9,9000.00,102,6);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (104,'Bruce','Ernst','bruce.
↪ernst@sqltutorial.org','590.423.4568','1991-05-21',9,6000.00,103,6);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (105,'David','Austin','david.
↪austin@sqltutorial.org','590.423.4569','1997-06-25',9,4800.00,103,6);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (106,'Valli','Pataballa','valli.
↪pataballa@sqltutorial.org','590.423.4560','1998-02-05',9,4800.00,103,6);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (107,'Diana','Lorentz','diana.
↪lorentz@sqltutorial.org','590.423.5567','1999-02-07',9,4200.00,103,6);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (108,'Nancy','Greenberg','nancy.
↪greenberg@sqltutorial.org','515.124.4569','1994-08-17',7,12000.00,101,10);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (109,'Daniel','Faviet','daniel.
↪faviet@sqltutorial.org','515.124.4169','1994-08-16',6,9000.00,108,10);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (110,'John','Chen','john.chen@sqltutorial.
↪org','515.124.4269','1997-09-28',6,8200.00,108,10);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (111,'Ismael','Sciarra','ismael.
↪sciarra@sqltutorial.org','515.124.4369','1997-09-30',6,7700.00,108,10);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (112,'Jose Manuel','Urman','jose manuel.
↪urman@sqltutorial.org','515.124.4469','1998-03-07',6,7800.00,108,10);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (113,'Luis','Popp','luis.popp@sqltutorial.
↪org','515.124.4567','1999-12-07',6,6900.00,108,10);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (114,'Den','Raphaely','den.
↪raphaely@sqltutorial.org','515.127.4561','1994-12-07',14,11000.00,100,3);
```

(continues on next page)

(continued from previous page)

```
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (115,'Alexander','Khoo','alexander.
↪khoo@sqltutorial.org','515.127.4562','1995-05-18',13,3100.00,114,3);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (116,'Shelli','Baida','shelli.
↪baida@sqltutorial.org','515.127.4563','1997-12-24',13,2900.00,114,3);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (117,'Sigal','Tobias','sigal.
↪tobias@sqltutorial.org','515.127.4564','1997-07-24',13,2800.00,114,3);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (118,'Guy','Himuro','guy.himuro@sqltutorial.
↪org','515.127.4565','1998-11-15',13,2600.00,114,3);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (119,'Karen','Colmenares','karen.
↪colmenares@sqltutorial.org','515.127.4566','1999-08-10',13,2500.00,114,3);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (120,'Matthew','Weiss','matthew.
↪weiss@sqltutorial.org','650.123.1234','1996-07-18',19,8000.00,100,5);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (121,'Adam','Fripp','adam.fripp@sqltutorial.
↪org','650.123.2234','1997-04-10',19,8200.00,100,5);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (122,'Payam','Kaufling','payam.
↪kaufling@sqltutorial.org','650.123.3234','1995-05-01',19,7900.00,100,5);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (123,'Shanta','Vollman','shanta.
↪vollman@sqltutorial.org','650.123.4234','1997-10-10',19,6500.00,100,5);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (126,'Irene','Mikkilineni','irene.
↪mikkilineni@sqltutorial.org','650.124.1224','1998-09-28',18,2700.00,120,5);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (145,'John','Russell','john.
↪russell@sqltutorial.org',NULL,'1996-10-01',15,14000.00,100,8);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (146,'Karen','Partners','karen.
↪partners@sqltutorial.org',NULL,'1997-01-05',15,13500.00,100,8);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (176,'Jonathon','Taylor','jonathon.
↪taylor@sqltutorial.org',NULL,'1998-03-24',16,8600.00,100,8);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (177,'Jack','Livingston','jack.
↪livingston@sqltutorial.org',NULL,'1998-04-23',16,8400.00,100,8);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (178,'Kimberely','Grant','kimberely.
↪grant@sqltutorial.org',NULL,'1999-05-24',16,7000.00,100,8);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (179,'Charles','Johnson','charles.
↪johnson@sqltutorial.org',NULL,'2000-01-04',16,6200.00,100,8);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (192,'Sarah','Bell','sarah.bell@sqltutorial.
↪org','650.501.1876','1996-02-04',17,4000.00,123,5);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (193,'Britney','Everett','britney.
↪everett@sqltutorial.org','650.501.2876','1997-03-03',17,3900.00,123,5);
```

(continues on next page)

(continued from previous page)

```
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (200,'Jennifer','Whalen','jennifer.
↪whalen@sqltutorial.org','515.123.4444','1987-09-17',3,4400.00,101,1);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (201,'Michael','Hartstein','michael.
↪hartstein@sqltutorial.org','515.123.5555','1996-02-17',10,13000.00,100,2);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (202,'Pat','Fay','pat.fay@sqltutorial.org',
↪'603.123.6666','1997-08-17',11,6000.00,201,2);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (203,'Susan','Mavris','susan.
↪mavris@sqltutorial.org','515.123.7777','1994-06-07',8,6500.00,101,4);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (204,'Hermann','Baer','hermann.
↪baer@sqltutorial.org','515.123.8888','1994-06-07',12,10000.00,101,7);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (205,'Shelley','Higgins','shelley.
↪higgins@sqltutorial.org','515.123.8080','1994-06-07',2,12000.00,101,11);
INSERT INTO employees(employee_id,first_name,last_name,email,phone_number,hire_date,job_
↪id,salary,manager_id,department_id) VALUES (206,'William','Gietz','william.
↪gietz@sqltutorial.org','515.123.8181','1994-06-07',1,8300.00,205,11);

/*Data for the table dependents */
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (1,'Penelope','Gietz','Child',206);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (2,'Nick','Higgins','Child',205);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (3,'Ed','Whalen','Child',200);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (4,'Jennifer','King','Child',100);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (5,'Johnny','Kochhar','Child',101);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (6,'Bette','De Haan','Child',102);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (7,'Grace','Faviet','Child',109);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (8,'Matthew','Chen','Child',110);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (9,'Joe','Sciarra','Child',111);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (10,'Christian','Urman','Child',112);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (11,'Zero','Popp','Child',113);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (12,'Karl','Greenberg','Child',108);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (13,'Uma','Mavris','Child',203);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (14,'Vivien','Hunold','Child',103);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (15,'Cuba','Ernst','Child',104);
```

(continues on next page)

(continued from previous page)

```
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (16,'Fred','Austin','Child',105);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (17,'Helen','Pataballa','Child',106);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (18,'Dan','Lorentz','Child',107);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (19,'Bob','Hartstein','Child',201);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (20,'Lucille','Fay','Child',202);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (21,'Kirsten','Baer','Child',204);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (22,'Elvis','Khoo','Child',115);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (23,'Sandra','Baida','Child',116);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (24,'Cameron','Tobias','Child',117);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (25,'Kevin','Himuro','Child',118);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (26,'Rip','Colmenares','Child',119);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (27,'Julia','Raphaely','Child',114);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (28,'Woody','Russell','Child',145);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (29,'Alec','Partners','Child',146);
INSERT INTO dependents(dependent_id,first_name,last_name,relationship,employee_id)␣
↪VALUES (30,'Sandra','Taylor','Child',176);
```

7.10.5 10.5 Querying Databases with SELECT

The most common SQL task is to retrieve data from one or more tables. The data is returned in the form of a result table, called *result set*. This is accomplished with the `SELECT` statement, which has the following general syntax.

```
SELECT
    column1, column2, columnN
FROM
    table_name
```

To run the code in this Jupyter Notebook, we will just insert the magic commands `%sql` or `%%sql` in front of each SQL code.

For example, in the next cell we retrieved the columns `employee_id, first_name, last_name, hire_date` from the `employees` table. When the statement is evaluated, the database system first evaluates the `FROM` clause and the `SELECT` clause afterward. I.e., from the table named `employees` select the listed columns.

[10]:
```
%%sql
SELECT
    employee_id, first_name, last_name, hire_date
```

(continues on next page)

(continued from previous page)

```
FROM
    employees
```

```
 * sqlite://
Done.
```

[10]:
```
[(100, 'Steven', 'King', '1987-06-17'),
 (101, 'Neena', 'Kochhar', '1989-09-21'),
 (102, 'Lex', 'De Haan', '1993-01-13'),
 (103, 'Alexander', 'Hunold', '1990-01-03'),
 (104, 'Bruce', 'Ernst', '1991-05-21'),
 (105, 'David', 'Austin', '1997-06-25'),
 (106, 'Valli', 'Pataballa', '1998-02-05'),
 (107, 'Diana', 'Lorentz', '1999-02-07'),
 (108, 'Nancy', 'Greenberg', '1994-08-17'),
 (109, 'Daniel', 'Faviet', '1994-08-16'),
 (110, 'John', 'Chen', '1997-09-28'),
 (111, 'Ismael', 'Sciarra', '1997-09-30'),
 (112, 'Jose Manuel', 'Urman', '1998-03-07'),
 (113, 'Luis', 'Popp', '1999-12-07'),
 (114, 'Den', 'Raphaely', '1994-12-07'),
 (115, 'Alexander', 'Khoo', '1995-05-18'),
 (116, 'Shelli', 'Baida', '1997-12-24'),
 (117, 'Sigal', 'Tobias', '1997-07-24'),
 (118, 'Guy', 'Himuro', '1998-11-15'),
 (119, 'Karen', 'Colmenares', '1999-08-10'),
 (120, 'Matthew', 'Weiss', '1996-07-18'),
 (121, 'Adam', 'Fripp', '1997-04-10'),
 (122, 'Payam', 'Kaufling', '1995-05-01'),
 (123, 'Shanta', 'Vollman', '1997-10-10'),
 (126, 'Irene', 'Mikkilineni', '1998-09-28'),
 (145, 'John', 'Russell', '1996-10-01'),
 (146, 'Karen', 'Partners', '1997-01-05'),
 (176, 'Jonathon', 'Taylor', '1998-03-24'),
 (177, 'Jack', 'Livingston', '1998-04-23'),
 (178, 'Kimberely', 'Grant', '1999-05-24'),
 (179, 'Charles', 'Johnson', '2000-01-04'),
 (192, 'Sarah', 'Bell', '1996-02-04'),
 (193, 'Britney', 'Everett', '1997-03-03'),
 (200, 'Jennifer', 'Whalen', '1987-09-17'),
 (201, 'Michael', 'Hartstein', '1996-02-17'),
 (202, 'Pat', 'Fay', '1997-08-17'),
 (203, 'Susan', 'Mavris', '1994-06-07'),
 (204, 'Hermann', 'Baer', '1994-06-07'),
 (205, 'Shelley', 'Higgins', '1994-06-07'),
 (206, 'William', 'Gietz', '1994-06-07')]
```

If we want to query all columns in a table we can use the asterisk operator * instead of the columns names.

[11]:
```
%sql SELECT * FROM employees
```

```
 * sqlite://
Done.
```

[11]:
```
[(100, 'Steven', 'King', 'steven.king@sqltutorial.org', '515.123.4567', '1987-06-17', 4,␣
↪24000.0, None, 9),
 (101, 'Neena', 'Kochhar', 'neena.kochhar@sqltutorial.org', '515.123.4568', '1989-09-21',
↪ 5, 17000.0, 100, 9),
 (102, 'Lex', 'De Haan', 'lex.de haan@sqltutorial.org', '515.123.4569', '1993-01-13', 5,␣
↪17000.0, 100, 9),
 (103, 'Alexander', 'Hunold', 'alexander.hunold@sqltutorial.org', '590.423.4567', '1990-
↪01-03', 9, 9000.0, 102, 6),
 (104, 'Bruce', 'Ernst', 'bruce.ernst@sqltutorial.org', '590.423.4568', '1991-05-21', 9,␣
↪6000.0, 103, 6),
 (105, 'David', 'Austin', 'david.austin@sqltutorial.org', '590.423.4569', '1997-06-25',␣
↪9, 4800.0, 103, 6),
 (106, 'Valli', 'Pataballa', 'valli.pataballa@sqltutorial.org', '590.423.4560', '1998-02-
↪05', 9, 4800.0, 103, 6),
 (107, 'Diana', 'Lorentz', 'diana.lorentz@sqltutorial.org', '590.423.5567', '1999-02-07',
↪ 9, 4200.0, 103, 6),
 (108, 'Nancy', 'Greenberg', 'nancy.greenberg@sqltutorial.org', '515.124.4569', '1994-08-
↪17', 7, 12000.0, 101, 10),
 (109, 'Daniel', 'Faviet', 'daniel.faviet@sqltutorial.org', '515.124.4169', '1994-08-16',
↪ 6, 9000.0, 108, 10),
 (110, 'John', 'Chen', 'john.chen@sqltutorial.org', '515.124.4269', '1997-09-28', 6,␣
↪8200.0, 108, 10),
 (111, 'Ismael', 'Sciarra', 'ismael.sciarra@sqltutorial.org', '515.124.4369', '1997-09-30
↪', 6, 7700.0, 108, 10),
 (112, 'Jose Manuel', 'Urman', 'jose manuel.urman@sqltutorial.org', '515.124.4469',
↪'1998-03-07', 6, 7800.0, 108, 10),
 (113, 'Luis', 'Popp', 'luis.popp@sqltutorial.org', '515.124.4567', '1999-12-07', 6,␣
↪6900.0, 108, 10),
 (114, 'Den', 'Raphaely', 'den.raphaely@sqltutorial.org', '515.127.4561', '1994-12-07',␣
↪14, 11000.0, 100, 3),
 (115, 'Alexander', 'Khoo', 'alexander.khoo@sqltutorial.org', '515.127.4562', '1995-05-18
↪', 13, 3100.0, 114, 3),
 (116, 'Shelli', 'Baida', 'shelli.baida@sqltutorial.org', '515.127.4563', '1997-12-24',␣
↪13, 2900.0, 114, 3),
 (117, 'Sigal', 'Tobias', 'sigal.tobias@sqltutorial.org', '515.127.4564', '1997-07-24',␣
↪13, 2800.0, 114, 3),
 (118, 'Guy', 'Himuro', 'guy.himuro@sqltutorial.org', '515.127.4565', '1998-11-15', 13,␣
↪2600.0, 114, 3),
 (119, 'Karen', 'Colmenares', 'karen.colmenares@sqltutorial.org', '515.127.4566', '1999-
↪08-10', 13, 2500.0, 114, 3),
 (120, 'Matthew', 'Weiss', 'matthew.weiss@sqltutorial.org', '650.123.1234', '1996-07-18',
↪ 19, 8000.0, 100, 5),
 (121, 'Adam', 'Fripp', 'adam.fripp@sqltutorial.org', '650.123.2234', '1997-04-10', 19,␣
↪8200.0, 100, 5),
 (122, 'Payam', 'Kaufling', 'payam.kaufling@sqltutorial.org', '650.123.3234', '1995-05-01
↪', 19, 7900.0, 100, 5),
 (123, 'Shanta', 'Vollman', 'shanta.vollman@sqltutorial.org', '650.123.4234', '1997-10-10
↪', 19, 6500.0, 100, 5),
 (126, 'Irene', 'Mikkilineni', 'irene.mikkilineni@sqltutorial.org', '650.124.1224',
↪'1998-09-28', 18, 2700.0, 120, 5),
 (145, 'John', 'Russell', 'john.russell@sqltutorial.org', None, '1996-10-01', 15, 14000.
↪0, 100, 8),
 (146, 'Karen', 'Partners', 'karen.partners@sqltutorial.org', None, '1997-01-05', 15,␣
↪13500.0, 100, 8),
```

(continues on next page)

(continued from previous page)

```
 (176, 'Jonathon', 'Taylor', 'jonathon.taylor@sqltutorial.org', None, '1998-03-24', 16,␣
↪8600.0, 100, 8),
 (177, 'Jack', 'Livingston', 'jack.livingston@sqltutorial.org', None, '1998-04-23', 16,␣
↪8400.0, 100, 8),
 (178, 'Kimberely', 'Grant', 'kimberely.grant@sqltutorial.org', None, '1999-05-24', 16,␣
↪7000.0, 100, 8),
 (179, 'Charles', 'Johnson', 'charles.johnson@sqltutorial.org', None, '2000-01-04', 16,␣
↪6200.0, 100, 8),
 (192, 'Sarah', 'Bell', 'sarah.bell@sqltutorial.org', '650.501.1876', '1996-02-04', 17,␣
↪4000.0, 123, 5),
 (193, 'Britney', 'Everett', 'britney.everett@sqltutorial.org', '650.501.2876', '1997-03-
↪03', 17, 3900.0, 123, 5),
 (200, 'Jennifer', 'Whalen', 'jennifer.whalen@sqltutorial.org', '515.123.4444', '1987-09-
↪17', 3, 4400.0, 101, 1),
 (201, 'Michael', 'Hartstein', 'michael.hartstein@sqltutorial.org', '515.123.5555',
↪'1996-02-17', 10, 13000.0, 100, 2),
 (202, 'Pat', 'Fay', 'pat.fay@sqltutorial.org', '603.123.6666', '1997-08-17', 11, 6000.0,
↪ 201, 2),
 (203, 'Susan', 'Mavris', 'susan.mavris@sqltutorial.org', '515.123.7777', '1994-06-07',␣
↪8, 6500.0, 101, 4),
 (204, 'Hermann', 'Baer', 'hermann.baer@sqltutorial.org', '515.123.8888', '1994-06-07',␣
↪12, 10000.0, 101, 7),
 (205, 'Shelley', 'Higgins', 'shelley.higgins@sqltutorial.org', '515.123.8080', '1994-06-
↪07', 2, 12000.0, 101, 11),
 (206, 'William', 'Gietz', 'william.gietz@sqltutorial.org', '515.123.8181', '1994-06-07',
↪ 1, 8300.0, 205, 11)]
```

List Tables in a Database

We can also use the `SELECT` statement to display a list of all tables in the current database. Every SQLite database has an `sqlite_master` table that defines the schema for the database. The following statement uses `sqlite_master` with the `type` field set to 'table' to return the names of all tables in the current database.

[12]:
```
%sql SELECT name FROM sqlite_master WHERE type='table'
```

```
 * sqlite://
Done.
```

[12]:
```
[('cars',),
 ('writer',),
 ('regions',),
 ('sqlite_sequence',),
 ('countries',),
 ('locations',),
 ('departments',),
 ('jobs',),
 ('employees',),
 ('dependents',)]
```

Here is the entire `sqlite_master` table for the current database. It stores metadata about the tables and other objects (e.g., indexes and views) in an SQLite database. It contains the columns `type`, `name`, `tbl_name`, `rootpage` (the root page of the object withing the SQL database file), and `sql` (the SQL CREATE statement that was used to create the database).

The `sqlite_master` table is automatically created and maintained by SQLite, and it is typically queried to inspect the schema of a database or retrieve information about its tables and other objects.

[13]:
```
%sql SELECT * FROM sqlite_master
```

```
 * sqlite://
Done.
```

[13]:
```
[('table', 'cars', 'cars', 2, 'CREATE TABLE cars(\n    id INTEGER NOT NULL PRIMARY KEY,\
↪n    name VARCHAR(50) NOT NULL,  \n    price INTEGER)'),
 ('table', 'writer', 'writer', 3, 'CREATE TABLE writer(\n    FirstName VARCHAR(50) NOT␣
↪NULL,\n    LastName VARCHAR(50) NOT NULL,  \n    Year INTEGER NOT NULL PRIMARY KEY)'),
 ('table', 'regions', 'regions', 4, 'CREATE TABLE regions (\n    region_id INTEGER␣
↪PRIMARY KEY AUTOINCREMENT NOT NULL,\n    region_name TEXT NOT NULL)'),
 ('table', 'sqlite_sequence', 'sqlite_sequence', 5, 'CREATE TABLE sqlite_sequence(name,
↪seq)'),
 ('table', 'countries', 'countries', 6, 'CREATE TABLE countries (\n    country_id TEXT␣
↪NOT NULL,\n    country_name TEXT NOT NULL,\n    region_id INTEGER NOT NULL,\n    ␣
↪PRIMARY KEY (country_id ASC),\n    FOREIGN KEY (region_id) REFERENCES regions (region_
↪id) ON DELETE CASCADE ON UPDATE CASCADE)'),
 ('index', 'sqlite_autoindex_countries_1', 'countries', 7, None),
 ('table', 'locations', 'locations', 8, 'CREATE TABLE locations (\n    location_id␣
↪INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,\n    street_address TEXT,\n    postal_code␣
↪TEXT,\n    city text ... (22 characters truncated) ... province TEXT,\n    country_id␣
↪INTEGER NOT NULL,\n    FOREIGN KEY (country_id) REFERENCES countries (country_id) ON␣
↪DELETE CASCADE ON UPDATE CASCADE)'),
 ('table', 'departments', 'departments', 9, 'CREATE TABLE departments (\n    department_
↪id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,\n    department_name TEXT NOT NULL,\n  ␣
↪ location_id INTEGER NOT NULL,\n    FOREIGN KEY (location_id) REFERENCES locations␣
↪(location_id) ON DELETE CASCADE ON UPDATE CASCADE)'),
 ('table', 'jobs', 'jobs', 10, 'CREATE TABLE jobs (\n    job_id INTEGER PRIMARY KEY␣
↪AUTOINCREMENT NOT NULL,\n    job_title TEXT NOT NULL,\n    min_salary DOUBLE NOT NULL,\
↪n    max_salary DOUBLE NOT NULL)'),
 ('table', 'employees', 'employees', 11, 'CREATE TABLE employees (\n    employee_id␣
↪INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,\n    first_name TEXT,\n    last_name TEXT␣
↪NOT NULL,\n    email  ... (339 characters truncated) ... rtment_id) ON DELETE CASCADE␣
↪ON UPDATE CASCADE,\n    FOREIGN KEY (manager_id) REFERENCES employees (employee_id) ON␣
↪DELETE CASCADE ON UPDATE CASCADE)'),
 ('table', 'dependents', 'dependents', 12, 'CREATE TABLE dependents (\n    dependent_id␣
↪INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,\n    first_name TEXT NOT NULL,\n    last_
↪name TEXT NOT NULL,\ ... (21 characters truncated) ... T NOT NULL,\n    employee_id␣
↪INTEGER NOT NULL,\n    FOREIGN KEY (employee_id) REFERENCES employees (employee_id) ON␣
↪DELETE CASCADE ON UPDATE CASCADE)')]
```

For instance, we can use the following code to retrieve the `CREATE` statement for the table `cars`.

[14]:
```
%sql SELECT sql FROM sqlite_master WHERE type='table' AND name='cars';
```

```
 * sqlite://
Done.
```

[14]:
```
[('CREATE TABLE cars(\n    id INTEGER NOT NULL PRIMARY KEY,\n    name VARCHAR(50) NOT␣
↪NULL,  \n    price INTEGER)',)]
```

Perform Simple Calculations in SELECT Statements

We can use standard math operators such as +, *, /, % in SELECT statements to perform simple mathematical calculations. The following expression creates a new column `salary * 1.05` from the `salary` column and adds 5% to the salary of every employee.

[15]:
```
%%sql
SELECT
    employee_id, first_name, salary, salary*1.05
FROM
    employees;
```

```
 * sqlite://
Done.
```

[15]:
```
[(100, 'Steven', 24000.0, 25200.0),
 (101, 'Neena', 17000.0, 17850.0),
 (102, 'Lex', 17000.0, 17850.0),
 (103, 'Alexander', 9000.0, 9450.0),
 (104, 'Bruce', 6000.0, 6300.0),
 (105, 'David', 4800.0, 5040.0),
 (106, 'Valli', 4800.0, 5040.0),
 (107, 'Diana', 4200.0, 4410.0),
 (108, 'Nancy', 12000.0, 12600.0),
 (109, 'Daniel', 9000.0, 9450.0),
 (110, 'John', 8200.0, 8610.0),
 (111, 'Ismael', 7700.0, 8085.0),
 (112, 'Jose Manuel', 7800.0, 8190.0),
 (113, 'Luis', 6900.0, 7245.0),
 (114, 'Den', 11000.0, 11550.0),
 (115, 'Alexander', 3100.0, 3255.0),
 (116, 'Shelli', 2900.0, 3045.0),
 (117, 'Sigal', 2800.0, 2940.0),
 (118, 'Guy', 2600.0, 2730.0),
 (119, 'Karen', 2500.0, 2625.0),
 (120, 'Matthew', 8000.0, 8400.0),
 (121, 'Adam', 8200.0, 8610.0),
 (122, 'Payam', 7900.0, 8295.0),
 (123, 'Shanta', 6500.0, 6825.0),
 (126, 'Irene', 2700.0, 2835.0),
 (145, 'John', 14000.0, 14700.0),
 (146, 'Karen', 13500.0, 14175.0),
 (176, 'Jonathon', 8600.0, 9030.0),
 (177, 'Jack', 8400.0, 8820.0),
 (178, 'Kimberely', 7000.0, 7350.0),
 (179, 'Charles', 6200.0, 6510.0),
 (192, 'Sarah', 4000.0, 4200.0),
 (193, 'Britney', 3900.0, 4095.0),
 (200, 'Jennifer', 4400.0, 4620.0),
 (201, 'Michael', 13000.0, 13650.0),
 (202, 'Pat', 6000.0, 6300.0),
 (203, 'Susan', 6500.0, 6825.0),
 (204, 'Hermann', 10000.0, 10500.0),
 (205, 'Shelley', 12000.0, 12600.0),
 (206, 'William', 8300.0, 8715.0)]
```

We can use `AS new_salary` to assign a different name for the newly created column.

[16]:
```
%%sql
SELECT
    employee_id, first_name, salary, salary*1.05 AS new_salary
FROM
    employees;
```

```
 * sqlite://
Done.
```

[16]:
```
[(100, 'Steven', 24000.0, 25200.0),
 (101, 'Neena', 17000.0, 17850.0),
 (102, 'Lex', 17000.0, 17850.0),
 (103, 'Alexander', 9000.0, 9450.0),
 (104, 'Bruce', 6000.0, 6300.0),
 (105, 'David', 4800.0, 5040.0),
 (106, 'Valli', 4800.0, 5040.0),
 (107, 'Diana', 4200.0, 4410.0),
 (108, 'Nancy', 12000.0, 12600.0),
 (109, 'Daniel', 9000.0, 9450.0),
 (110, 'John', 8200.0, 8610.0),
 (111, 'Ismael', 7700.0, 8085.0),
 (112, 'Jose Manuel', 7800.0, 8190.0),
 (113, 'Luis', 6900.0, 7245.0),
 (114, 'Den', 11000.0, 11550.0),
 (115, 'Alexander', 3100.0, 3255.0),
 (116, 'Shelli', 2900.0, 3045.0),
 (117, 'Sigal', 2800.0, 2940.0),
 (118, 'Guy', 2600.0, 2730.0),
 (119, 'Karen', 2500.0, 2625.0),
 (120, 'Matthew', 8000.0, 8400.0),
 (121, 'Adam', 8200.0, 8610.0),
 (122, 'Payam', 7900.0, 8295.0),
 (123, 'Shanta', 6500.0, 6825.0),
 (126, 'Irene', 2700.0, 2835.0),
 (145, 'John', 14000.0, 14700.0),
 (146, 'Karen', 13500.0, 14175.0),
 (176, 'Jonathon', 8600.0, 9030.0),
 (177, 'Jack', 8400.0, 8820.0),
 (178, 'Kimberely', 7000.0, 7350.0),
 (179, 'Charles', 6200.0, 6510.0),
 (192, 'Sarah', 4000.0, 4200.0),
 (193, 'Britney', 3900.0, 4095.0),
 (200, 'Jennifer', 4400.0, 4620.0),
 (201, 'Michael', 13000.0, 13650.0),
 (202, 'Pat', 6000.0, 6300.0),
 (203, 'Susan', 6500.0, 6825.0),
 (204, 'Hermann', 10000.0, 10500.0),
 (205, 'Shelley', 12000.0, 12600.0),
 (206, 'William', 8300.0, 8715.0)]
```

7.10.6 10.6 Sorting Data with ORDER BY

The clause `ORDER BY` can be used within a `SELECT` statement to sort the returned rows.

The general syntax is:

```
SELECT
    column1, column2, columnN
FROM
    table_name
ORDER BY sort_expression [ASC | DESC];
```

The `sort_expression` specifies the sort criteria, whereas `ASC` or `DESC` indicates whether to sort the result set into ascending (default) or descending order.

The following example uses the clause `ORDER BY` to sort employees by first names in alphabetical order.

[17]:
```
%%sql
SELECT
    employee_id, first_name, last_name, hire_date, salary
FROM
    employees
ORDER BY first_name;
```

```
 * sqlite://
Done.
```

[17]:
```
[(121, 'Adam', 'Fripp', '1997-04-10', 8200.0),
 (103, 'Alexander', 'Hunold', '1990-01-03', 9000.0),
 (115, 'Alexander', 'Khoo', '1995-05-18', 3100.0),
 (193, 'Britney', 'Everett', '1997-03-03', 3900.0),
 (104, 'Bruce', 'Ernst', '1991-05-21', 6000.0),
 (179, 'Charles', 'Johnson', '2000-01-04', 6200.0),
 (109, 'Daniel', 'Faviet', '1994-08-16', 9000.0),
 (105, 'David', 'Austin', '1997-06-25', 4800.0),
 (114, 'Den', 'Raphaely', '1994-12-07', 11000.0),
 (107, 'Diana', 'Lorentz', '1999-02-07', 4200.0),
 (118, 'Guy', 'Himuro', '1998-11-15', 2600.0),
 (204, 'Hermann', 'Baer', '1994-06-07', 10000.0),
 (126, 'Irene', 'Mikkilineni', '1998-09-28', 2700.0),
 (111, 'Ismael', 'Sciarra', '1997-09-30', 7700.0),
 (177, 'Jack', 'Livingston', '1998-04-23', 8400.0),
 (200, 'Jennifer', 'Whalen', '1987-09-17', 4400.0),
 (110, 'John', 'Chen', '1997-09-28', 8200.0),
 (145, 'John', 'Russell', '1996-10-01', 14000.0),
 (176, 'Jonathon', 'Taylor', '1998-03-24', 8600.0),
 (112, 'Jose Manuel', 'Urman', '1998-03-07', 7800.0),
 (119, 'Karen', 'Colmenares', '1999-08-10', 2500.0),
 (146, 'Karen', 'Partners', '1997-01-05', 13500.0),
 (178, 'Kimberely', 'Grant', '1999-05-24', 7000.0),
 (102, 'Lex', 'De Haan', '1993-01-13', 17000.0),
 (113, 'Luis', 'Popp', '1999-12-07', 6900.0),
 (120, 'Matthew', 'Weiss', '1996-07-18', 8000.0),
 (201, 'Michael', 'Hartstein', '1996-02-17', 13000.0),
 (108, 'Nancy', 'Greenberg', '1994-08-17', 12000.0),
 (101, 'Neena', 'Kochhar', '1989-09-21', 17000.0),
```

(continues on next page)

(continued from previous page)

```
 (202, 'Pat', 'Fay', '1997-08-17', 6000.0),
 (122, 'Payam', 'Kaufling', '1995-05-01', 7900.0),
 (192, 'Sarah', 'Bell', '1996-02-04', 4000.0),
 (123, 'Shanta', 'Vollman', '1997-10-10', 6500.0),
 (205, 'Shelley', 'Higgins', '1994-06-07', 12000.0),
 (116, 'Shelli', 'Baida', '1997-12-24', 2900.0),
 (117, 'Sigal', 'Tobias', '1997-07-24', 2800.0),
 (100, 'Steven', 'King', '1987-06-17', 24000.0),
 (203, 'Susan', 'Mavris', '1994-06-07', 6500.0),
 (106, 'Valli', 'Pataballa', '1998-02-05', 4800.0),
 (206, 'William', 'Gietz', '1994-06-07', 8300.0)]
```

The `ORDER BY` clause also allows using multiple expressions for sorting, separated by commas. In the following example `ORDER BY` is used to sort the employees by their first name in ascending order, and the employees who have the same first name are further sorted by their last name in descending order. E.g., check the sorting for the two employees with the name Alexander.

[18]:
```
%%sql
SELECT
    employee_id, first_name, last_name, hire_date, salary
FROM
    employees
ORDER BY first_name, last_name DESC;
```

```
 * sqlite://
Done.
```

[18]:
```
[(121, 'Adam', 'Fripp', '1997-04-10', 8200.0),
 (115, 'Alexander', 'Khoo', '1995-05-18', 3100.0),
 (103, 'Alexander', 'Hunold', '1990-01-03', 9000.0),
 (193, 'Britney', 'Everett', '1997-03-03', 3900.0),
 (104, 'Bruce', 'Ernst', '1991-05-21', 6000.0),
 (179, 'Charles', 'Johnson', '2000-01-04', 6200.0),
 (109, 'Daniel', 'Faviet', '1994-08-16', 9000.0),
 (105, 'David', 'Austin', '1997-06-25', 4800.0),
 (114, 'Den', 'Raphaely', '1994-12-07', 11000.0),
 (107, 'Diana', 'Lorentz', '1999-02-07', 4200.0),
 (118, 'Guy', 'Himuro', '1998-11-15', 2600.0),
 (204, 'Hermann', 'Baer', '1994-06-07', 10000.0),
 (126, 'Irene', 'Mikkilineni', '1998-09-28', 2700.0),
 (111, 'Ismael', 'Sciarra', '1997-09-30', 7700.0),
 (177, 'Jack', 'Livingston', '1998-04-23', 8400.0),
 (200, 'Jennifer', 'Whalen', '1987-09-17', 4400.0),
 (145, 'John', 'Russell', '1996-10-01', 14000.0),
 (110, 'John', 'Chen', '1997-09-28', 8200.0),
 (176, 'Jonathon', 'Taylor', '1998-03-24', 8600.0),
 (112, 'Jose Manuel', 'Urman', '1998-03-07', 7800.0),
 (146, 'Karen', 'Partners', '1997-01-05', 13500.0),
 (119, 'Karen', 'Colmenares', '1999-08-10', 2500.0),
 (178, 'Kimberely', 'Grant', '1999-05-24', 7000.0),
 (102, 'Lex', 'De Haan', '1993-01-13', 17000.0),
 (113, 'Luis', 'Popp', '1999-12-07', 6900.0),
 (120, 'Matthew', 'Weiss', '1996-07-18', 8000.0),
```

(continues on next page)

(continued from previous page)

```
 (201, 'Michael', 'Hartstein', '1996-02-17', 13000.0),
 (108, 'Nancy', 'Greenberg', '1994-08-17', 12000.0),
 (101, 'Neena', 'Kochhar', '1989-09-21', 17000.0),
 (202, 'Pat', 'Fay', '1997-08-17', 6000.0),
 (122, 'Payam', 'Kaufling', '1995-05-01', 7900.0),
 (192, 'Sarah', 'Bell', '1996-02-04', 4000.0),
 (123, 'Shanta', 'Vollman', '1997-10-10', 6500.0),
 (205, 'Shelley', 'Higgins', '1994-06-07', 12000.0),
 (116, 'Shelli', 'Baida', '1997-12-24', 2900.0),
 (117, 'Sigal', 'Tobias', '1997-07-24', 2800.0),
 (100, 'Steven', 'King', '1987-06-17', 24000.0),
 (203, 'Susan', 'Mavris', '1994-06-07', 6500.0),
 (106, 'Valli', 'Pataballa', '1998-02-05', 4800.0),
 (206, 'William', 'Gietz', '1994-06-07', 8300.0)]
```

Similarly, we can use `ORDER BY` to sort columns with numerical data, or to sort by date as in the following cell.

[19]:
```
%%sql
SELECT
    employee_id, first_name, last_name, hire_date, salary
FROM
    employees
ORDER BY hire_date;
```

```
 * sqlite://
Done.
```

[19]:
```
[(100, 'Steven', 'King', '1987-06-17', 24000.0),
 (200, 'Jennifer', 'Whalen', '1987-09-17', 4400.0),
 (101, 'Neena', 'Kochhar', '1989-09-21', 17000.0),
 (103, 'Alexander', 'Hunold', '1990-01-03', 9000.0),
 (104, 'Bruce', 'Ernst', '1991-05-21', 6000.0),
 (102, 'Lex', 'De Haan', '1993-01-13', 17000.0),
 (203, 'Susan', 'Mavris', '1994-06-07', 6500.0),
 (204, 'Hermann', 'Baer', '1994-06-07', 10000.0),
 (205, 'Shelley', 'Higgins', '1994-06-07', 12000.0),
 (206, 'William', 'Gietz', '1994-06-07', 8300.0),
 (109, 'Daniel', 'Faviet', '1994-08-16', 9000.0),
 (108, 'Nancy', 'Greenberg', '1994-08-17', 12000.0),
 (114, 'Den', 'Raphaely', '1994-12-07', 11000.0),
 (122, 'Payam', 'Kaufling', '1995-05-01', 7900.0),
 (115, 'Alexander', 'Khoo', '1995-05-18', 3100.0),
 (192, 'Sarah', 'Bell', '1996-02-04', 4000.0),
 (201, 'Michael', 'Hartstein', '1996-02-17', 13000.0),
 (120, 'Matthew', 'Weiss', '1996-07-18', 8000.0),
 (145, 'John', 'Russell', '1996-10-01', 14000.0),
 (146, 'Karen', 'Partners', '1997-01-05', 13500.0),
 (193, 'Britney', 'Everett', '1997-03-03', 3900.0),
 (121, 'Adam', 'Fripp', '1997-04-10', 8200.0),
 (105, 'David', 'Austin', '1997-06-25', 4800.0),
 (117, 'Sigal', 'Tobias', '1997-07-24', 2800.0),
 (202, 'Pat', 'Fay', '1997-08-17', 6000.0),
 (110, 'John', 'Chen', '1997-09-28', 8200.0),
```

(continues on next page)

(continued from previous page)

```
 (111, 'Ismael', 'Sciarra', '1997-09-30', 7700.0),
 (123, 'Shanta', 'Vollman', '1997-10-10', 6500.0),
 (116, 'Shelli', 'Baida', '1997-12-24', 2900.0),
 (106, 'Valli', 'Pataballa', '1998-02-05', 4800.0),
 (112, 'Jose Manuel', 'Urman', '1998-03-07', 7800.0),
 (176, 'Jonathon', 'Taylor', '1998-03-24', 8600.0),
 (177, 'Jack', 'Livingston', '1998-04-23', 8400.0),
 (126, 'Irene', 'Mikkilineni', '1998-09-28', 2700.0),
 (118, 'Guy', 'Himuro', '1998-11-15', 2600.0),
 (107, 'Diana', 'Lorentz', '1999-02-07', 4200.0),
 (178, 'Kimberely', 'Grant', '1999-05-24', 7000.0),
 (119, 'Karen', 'Colmenares', '1999-08-10', 2500.0),
 (113, 'Luis', 'Popp', '1999-12-07', 6900.0),
 (179, 'Charles', 'Johnson', '2000-01-04', 6200.0)]
```

7.10.7 10.7 Filtering Data

LIMIT

`LIMIT` is used to constrain the number of rows returned by a query, similar to the functions `head()` and `tail()` in Pandas.

[20]:
```
%%sql
SELECT
    employee_id, first_name, last_name, hire_date, salary
FROM
    employees
ORDER BY first_name
LIMIT 5;
```

```
 * sqlite://
Done.
```

[20]:
```
[(121, 'Adam', 'Fripp', '1997-04-10', 8200.0),
 (103, 'Alexander', 'Hunold', '1990-01-03', 9000.0),
 (115, 'Alexander', 'Khoo', '1995-05-18', 3100.0),
 (193, 'Britney', 'Everett', '1997-03-03', 3900.0),
 (104, 'Bruce', 'Ernst', '1991-05-21', 6000.0)]
```

It is also possible to include an `OFFSET` clause, which will skip rows before retrieving the data. E.g., in the next cell, the first 3 rows are skipped, and rows 4-8 are returned.

[21]:
```
%%sql
SELECT
    employee_id, first_name, last_name, hire_date, salary
FROM
    employees
ORDER BY first_name
LIMIT 5
OFFSET 3;
```

```
 * sqlite://
Done.
```

[21]:
```
[(193, 'Britney', 'Everett', '1997-03-03', 3900.0),
 (104, 'Bruce', 'Ernst', '1991-05-21', 6000.0),
 (179, 'Charles', 'Johnson', '2000-01-04', 6200.0),
 (109, 'Daniel', 'Faviet', '1994-08-16', 9000.0),
 (105, 'David', 'Austin', '1997-06-25', 4800.0)]
```

DISTINCT

The `DISTINCT` clause allows to remove duplicate rows from the result set.

E.g., the next cell shows the first 15 rows of the salary columns, where some rows have the same value. In the cell afterward, `DISTINCT` is used to remove the rows with the same values for salary.

[22]:
```
%%sql
SELECT
    salary
FROM
    employees
ORDER BY salary DESC
LIMIT 15;
```

```
 * sqlite://
Done.
```

[22]:
```
[(24000.0,),
 (17000.0,),
 (17000.0,),
 (14000.0,),
 (13500.0,),
 (13000.0,),
 (12000.0,),
 (12000.0,),
 (11000.0,),
 (10000.0,),
 (9000.0,),
 (9000.0,),
 (8600.0,),
 (8400.0,),
 (8300.0,)]
```

[23]:
```
%%sql
SELECT DISTINCT
    salary
FROM
    employees
ORDER BY salary DESC
LIMIT 15;
```

```
 * sqlite://
Done.
```

[23]:
```
[(24000.0,),
 (17000.0,),
 (14000.0,),
```

(continues on next page)

(continued from previous page)

```
 (13500.0,),
 (13000.0,),
 (12000.0,),
 (11000.0,),
 (10000.0,),
 (9000.0,),
 (8600.0,),
 (8400.0,),
 (8300.0,),
 (8200.0,),
 (8000.0,),
 (7900.0,)]
```

WHERE

The `WHERE` clause filters data based on specified conditions. For instance, return only the employees that have a salary greater than a certain value.

[24]:
```
%%sql
SELECT DISTINCT
    employee_id, first_name, last_name,salary
FROM
    employees
WHERE salary > 9000
ORDER BY salary DESC;
```

```
 * sqlite://
Done.
```

[24]:
```
[(100, 'Steven', 'King', 24000.0),
 (101, 'Neena', 'Kochhar', 17000.0),
 (102, 'Lex', 'De Haan', 17000.0),
 (145, 'John', 'Russell', 14000.0),
 (146, 'Karen', 'Partners', 13500.0),
 (201, 'Michael', 'Hartstein', 13000.0),
 (108, 'Nancy', 'Greenberg', 12000.0),
 (205, 'Shelley', 'Higgins', 12000.0),
 (114, 'Den', 'Raphaely', 11000.0),
 (204, 'Hermann', 'Baer', 10000.0)]
```

Or, return the employees who work in the department 5.

[25]:
```
%%sql
SELECT DISTINCT
    employee_id, first_name, last_name, salary, department_id
FROM
    employees
WHERE department_id = 5
ORDER BY first_name;
```

```
 * sqlite://
Done.
```

[25]:
```
[(121, 'Adam', 'Fripp', 8200.0, 5),
 (193, 'Britney', 'Everett', 3900.0, 5),
 (126, 'Irene', 'Mikkilineni', 2700.0, 5),
 (120, 'Matthew', 'Weiss', 8000.0, 5),
 (122, 'Payam', 'Kaufling', 7900.0, 5),
 (192, 'Sarah', 'Bell', 4000.0, 5),
 (123, 'Shanta', 'Vollman', 6500.0, 5)]
```

Comparison Operators

To specify a condition, we can use the standard comparison operators, such as >, <, >=, <=, =, and note that <> can be used for 'not equal to'.

[26]:
```
%%sql
SELECT DISTINCT
    employee_id, first_name, last_name, salary, department_id
FROM
    employees
WHERE department_id <> 5
ORDER BY first_name;
```

```
 * sqlite://
Done.
```

[26]:
```
[(103, 'Alexander', 'Hunold', 9000.0, 6),
 (115, 'Alexander', 'Khoo', 3100.0, 3),
 (104, 'Bruce', 'Ernst', 6000.0, 6),
 (179, 'Charles', 'Johnson', 6200.0, 8),
 (109, 'Daniel', 'Faviet', 9000.0, 10),
 (105, 'David', 'Austin', 4800.0, 6),
 (114, 'Den', 'Raphaely', 11000.0, 3),
 (107, 'Diana', 'Lorentz', 4200.0, 6),
 (118, 'Guy', 'Himuro', 2600.0, 3),
 (204, 'Hermann', 'Baer', 10000.0, 7),
 (111, 'Ismael', 'Sciarra', 7700.0, 10),
 (177, 'Jack', 'Livingston', 8400.0, 8),
 (200, 'Jennifer', 'Whalen', 4400.0, 1),
 (110, 'John', 'Chen', 8200.0, 10),
 (145, 'John', 'Russell', 14000.0, 8),
 (176, 'Jonathon', 'Taylor', 8600.0, 8),
 (112, 'Jose Manuel', 'Urman', 7800.0, 10),
 (119, 'Karen', 'Colmenares', 2500.0, 3),
 (146, 'Karen', 'Partners', 13500.0, 8),
 (178, 'Kimberely', 'Grant', 7000.0, 8),
 (102, 'Lex', 'De Haan', 17000.0, 9),
 (113, 'Luis', 'Popp', 6900.0, 10),
 (201, 'Michael', 'Hartstein', 13000.0, 2),
 (108, 'Nancy', 'Greenberg', 12000.0, 10),
 (101, 'Neena', 'Kochhar', 17000.0, 9),
 (202, 'Pat', 'Fay', 6000.0, 2),
 (205, 'Shelley', 'Higgins', 12000.0, 11),
 (116, 'Shelli', 'Baida', 2900.0, 3),
 (117, 'Sigal', 'Tobias', 2800.0, 3),
```

(continues on next page)

(continued from previous page)

```
(100, 'Steven', 'King', 24000.0, 9),
(203, 'Susan', 'Mavris', 6500.0, 4),
(106, 'Valli', 'Pataballa', 4800.0, 6),
(206, 'William', 'Gietz', 8300.0, 11)]
```

Logical Operators

We can also use logical operators to combine multiple conditions in the `WHERE` clause of an SQL statement. The following table contains the SQL logical operators.

Operator	Meaning
ALL	Return true if all comparisons are true
AND	Return true if both expressions are true
ANY	Return true if any one of the comparisons is true.
BETWEEN	Return true if the operand is within a range
EXISTS	Return true if a subquery contains any rows
IN	Return true if the operand is equal to one of the value in a list
LIKE	Return true if the operand matches a pattern
NOT	Reverse the result of any other Boolean operator.
OR	Return true if either expression is true
SOME	Return true if some of the expressions are true

Figure source: Reference [1].

[27]:
```
%%sql
SELECT
    first_name, last_name, salary
FROM
    employees
WHERE salary > 5000 AND salary < 7000
ORDER BY salary;
```

```
 * sqlite://
Done.
```

[27]:
```
[('Bruce', 'Ernst', 6000.0),
 ('Pat', 'Fay', 6000.0),
 ('Charles', 'Johnson', 6200.0),
 ('Shanta', 'Vollman', 6500.0),
 ('Susan', 'Mavris', 6500.0),
 ('Luis', 'Popp', 6900.0)]
```

[28]:
```
%%sql
SELECT
    first_name, last_name, salary
FROM
    employees
WHERE salary BETWEEN 5000 AND 7000
ORDER BY salary;
```

```
 * sqlite://
Done.
```

[28]:
```
[('Bruce', 'Ernst', 6000.0),
 ('Pat', 'Fay', 6000.0),
 ('Charles', 'Johnson', 6200.0),
 ('Shanta', 'Vollman', 6500.0),
 ('Susan', 'Mavris', 6500.0),
 ('Luis', 'Popp', 6900.0),
 ('Kimberely', 'Grant', 7000.0)]
```

[29]:
```
%%sql
SELECT
    first_name, last_name, salary
FROM
    employees
WHERE salary = 6000 OR salary = 7000
ORDER BY salary;
```

```
 * sqlite://
Done.
```

[29]:
```
[('Bruce', 'Ernst', 6000.0),
 ('Pat', 'Fay', 6000.0),
 ('Kimberely', 'Grant', 7000.0)]
```

Select employees with first names that start with `jo`.

[30]:
```
%%sql
SELECT
    employee_id, first_name, last_name
FROM
    employees
WHERE first_name LIKE 'jo%'
ORDER BY first_name;
```

```
 * sqlite://
Done.
```

[30]:
```
[(110, 'John', 'Chen'),
 (145, 'John', 'Russell'),
 (176, 'Jonathon', 'Taylor'),
 (112, 'Jose Manuel', 'Urman')]
```

Select employees with first names whose second character is h.

[31]:
```
%%sql
SELECT
    employee_id, first_name, last_name
FROM
    employees
WHERE
    first_name LIKE '_h%'
ORDER BY first_name;
```

```
 * sqlite://
Done.
```

[31]:
```
[(179, 'Charles', 'Johnson'),
 (123, 'Shanta', 'Vollman'),
 (205, 'Shelley', 'Higgins'),
 (116, 'Shelli', 'Baida')]
```

[32]:
```
%%sql
SELECT
    first_name, last_name, department_id
FROM
    employees
WHERE department_id IN (8, 9)
ORDER BY department_id;
```

```
 * sqlite://
Done.
```

[32]:
```
[('John', 'Russell', 8),
 ('Karen', 'Partners', 8),
 ('Jonathon', 'Taylor', 8),
 ('Jack', 'Livingston', 8),
 ('Kimberely', 'Grant', 8),
 ('Charles', 'Johnson', 8),
 ('Steven', 'King', 9),
 ('Neena', 'Kochhar', 9),
 ('Lex', 'De Haan', 9)]
```

IS NULL Operator

To determine whether a row or column has missing or non-defined values, we can use the `IS NULL` operator.

For instance, to find all employees who do not have phone numbers, we use the `IS NUL` operator as follows.

[33]:
```
%%sql
SELECT
    first_name, last_name, phone_number
FROM
    employees
WHERE phone_number IS NULL
ORDER BY first_name , last_name;
```

```
 * sqlite://
Done.
```

[33]:
```
[('Charles', 'Johnson', None),
 ('Jack', 'Livingston', None),
 ('John', 'Russell', None),
 ('Jonathon', 'Taylor', None),
 ('Karen', 'Partners', None),
 ('Kimberely', 'Grant', None)]
```

7.10.8 10.8 Conditional Expressions

The `CASE` expression is used to add if-then-else logic to SQL statements, which allows to evaluate a list of conditions and returns one of the possible results.

In the next cell, the `CASE` expression returns the results `Low`, `Average`, or `High` based on the conditions regarding the salary. The results are collected in the `evaluation` column.

[34]:
```
%%sql
SELECT
    first_name, last_name,
CASE WHEN salary < 3000 THEN 'Low'
     WHEN salary >= 3000 AND salary <= 5000 THEN 'Average'
     WHEN salary > 5000 THEN 'High'
END evaluation
FROM
    employees
LIMIT 15;
```

```
 * sqlite://
Done.
```

[34]:
```
[('Steven', 'King', 'High'),
 ('Neena', 'Kochhar', 'High'),
 ('Lex', 'De Haan', 'High'),
 ('Alexander', 'Hunold', 'High'),
 ('Bruce', 'Ernst', 'High'),
 ('David', 'Austin', 'Average'),
 ('Valli', 'Pataballa', 'Average'),
 ('Diana', 'Lorentz', 'Average'),
 ('Nancy', 'Greenberg', 'High'),
```

(continues on next page)

(continued from previous page)

```
 ('Daniel', 'Faviet', 'High'),
 ('John', 'Chen', 'High'),
 ('Ismael', 'Sciarra', 'High'),
 ('Jose Manuel', 'Urman', 'High'),
 ('Luis', 'Popp', 'High'),
 ('Den', 'Raphaely', 'High')]
```

7.10.9 10.9 Joining Multiple Tables

SQL provides several ways to join tables, such as `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `FULL JOIN`, `CROSS JOIN`, and others.

INNER JOIN

`INNER JOIN` combines rows from two or more tables based on a related column between them. It retrieves only the rows that have matching values in both tables.

The syntax is:

```
SELECT
    column1, column2, ...
FROM
    table1
INNER JOIN table2 ON table1.columnX = table2.columnX;
```

It specifies to join `table1` and `table2` by matching values in `columnX` from `table1` and `columnX` from `table2`, adn returns `column1, column2, ...` from the joined tables.

Let's show how we can use `INNER JOIN`, where for instance, we want to retrieve the list of employees who work in departments 1, 2, and 3, and we would like to list the names of the departments.

To do that, first we can notice that both the `employees` table and the `departments` table have a column `department_id`. Therefore, we can use the `department_id` column in the `employees` table as the foreign key column to link the `employees` to the `departments` table.

Let's first display the employees who work in departments 1, 2, and 3.

[35]:
```
%%sql
SELECT
    first_name, last_name, department_id
FROM
    employees
WHERE department_id IN (1, 2, 3)
ORDER BY department_id;
```

```
 * sqlite://
Done.
```

[35]:
```
[('Jennifer', 'Whalen', 1),
 ('Michael', 'Hartstein', 2),
 ('Pat', 'Fay', 2),
 ('Den', 'Raphaely', 3),
 ('Alexander', 'Khoo', 3),
```

(continues on next page)

(continued from previous page)

```
 ('Shelli', 'Baida', 3),
 ('Sigal', 'Tobias', 3),
 ('Guy', 'Himuro', 3),
 ('Karen', 'Colmenares', 3)]
```

We can also find the names of the departments that have a `department_id` of 1, 2, and 3.

[36]:
```
%%sql
SELECT
    department_id, department_name
FROM
    departments
WHERE department_id IN (1, 2, 3);
```

```
 * sqlite://
Done.
```

[36]:
```
[(1, 'Administration'), (2, 'Marketing'), (3, 'Purchasing')]
```

Next, we join the `employees` and `departments` tables, and use `INNER JOIN` to match the `department_id` column in the two tables. For each row, if the condition `departments.department_id = employees.department_id` is satisfied, the combined row will include data from rows in both `employees` and `departments` tables in the result set.

[37]:
```
%%sql
SELECT
    first_name, last_name, employees.department_id, departments.department_id,␣
↪department_name
FROM
    employees
INNER JOIN departments ON departments.department_id = employees.department_id
WHERE employees.department_id IN (1, 2, 3);
```

```
 * sqlite://
Done.
```

[37]:
```
[('Jennifer', 'Whalen', 1, 1, 'Administration'),
 ('Michael', 'Hartstein', 2, 2, 'Marketing'),
 ('Pat', 'Fay', 2, 2, 'Marketing'),
 ('Den', 'Raphaely', 3, 3, 'Purchasing'),
 ('Alexander', 'Khoo', 3, 3, 'Purchasing'),
 ('Shelli', 'Baida', 3, 3, 'Purchasing'),
 ('Sigal', 'Tobias', 3, 3, 'Purchasing'),
 ('Guy', 'Himuro', 3, 3, 'Purchasing'),
 ('Karen', 'Colmenares', 3, 3, 'Purchasing')]
```

Let's look at one more example, where we used `INNER JOIN` to join three tables. Specifically, in the next cell we use one more `INNER JOIN` clause to join the tables `employees` and `jobs` using the `jobs_id` column, in order to retrieve the information about the `job_title` column from the `jobs` table.

[38]:
```
%%sql
SELECT
    first_name, last_name, job_title, department_name
FROM
    employees
INNER JOIN departments ON departments.department_id = employees.department_id
```

(continues on next page)

(continued from previous page)

```
INNER JOIN jobs ON jobs.job_id = employees.job_id
WHERE employees.department_id IN (1, 2, 3);
```

```
 * sqlite://
Done.
```

[38]:
```
[('Den', 'Raphaely', 'Purchasing Manager', 'Purchasing'),
 ('Alexander', 'Khoo', 'Purchasing Clerk', 'Purchasing'),
 ('Shelli', 'Baida', 'Purchasing Clerk', 'Purchasing'),
 ('Sigal', 'Tobias', 'Purchasing Clerk', 'Purchasing'),
 ('Guy', 'Himuro', 'Purchasing Clerk', 'Purchasing'),
 ('Karen', 'Colmenares', 'Purchasing Clerk', 'Purchasing'),
 ('Jennifer', 'Whalen', 'Administration Assistant', 'Administration'),
 ('Michael', 'Hartstein', 'Marketing Manager', 'Marketing'),
 ('Pat', 'Fay', 'Marketing Representative', 'Marketing')]
```

LEFT JOIN

`LEFT JOIN` (also referred to as `LEFT OUTER JOIN`) is similar to `INNER JOIN`, only it retrieves all rows from the left table in the `JOIN` clause and the matched rows from the right table.

The "left table" is the table that appears on the left side of the `JOIN` clause. It is considered the primary table, and its rows are the ones that we want to retrieve or work with primarily.

The "right table" is the table that appears on the right side of the `JOIN` clause. It is the secondary table, and we are typically interested in its data to complement or match with the data from the left table.

For instance, the `countries` table is shown below, with `country_id` column being the primary key. The following query returns columns for the countries US, UK, and China.

[39]:
```
%%sql
SELECT
    country_id,
    country_name
FROM
    countries
WHERE country_id IN ('US', 'UK', 'CN');
```

```
 * sqlite://
Done.
```

[39]:
```
[('CN', 'China'), ('UK', 'United Kingdom'), ('US', 'United States of America')]
```

Similarly, the `locations` table shown below has a `country_id` column as the foreign key. The query returns the locations located in the US, UK, and China.

[40]:
```
%%sql
SELECT
    country_id,
    street_address,
    city
FROM
    locations
WHERE country_id IN ('US', 'UK', 'CN');
```

```
 * sqlite://
Done.
```

[40]:
```
[('US', '2014 Jabberwocky Rd', 'Southlake'),
 ('US', '2011 Interiors Blvd', 'South San Francisco'),
 ('US', '2004 Charade Rd', 'Seattle'),
 ('UK', '8204 Arthur St', 'London'),
 ('UK', 'Magdalen Centre, The Oxford Science Park', 'Oxford')]
```

In the next cell, because we use the `LEFT JOIN` clause, all rows that satisfy the condition in the `WHERE` clause of the `countries` table are included in the result set.

For each row in the `countries` table, the `LEFT JOIN` clause finds the matching rows in the `locations` table. If at least one matching row is found, the database engine combines the data from columns of the matching rows in both tables.

In this case, there is no matching row found for the country with the `country_id` of CN, and therefore, the row in the `countries` table is included in the result set and the row in the `locations` table is filled with `None` values.

[41]:
```
%%sql
SELECT
    countries.country_name,
    countries.country_id,
    locations.country_id,
    locations.street_address,
    locations.city
FROM
    countries
LEFT JOIN locations ON locations.country_id = countries.country_id
WHERE countries.country_id IN ('US', 'UK', 'CN')
```

```
 * sqlite://
Done.
```

[41]:
```
[('China', 'CN', None, None, None),
 ('United Kingdom', 'UK', 'UK', '8204 Arthur St', 'London'),
 ('United Kingdom', 'UK', 'UK', 'Magdalen Centre, The Oxford Science Park', 'Oxford'),
 ('United States of America', 'US', 'US', '2014 Jabberwocky Rd', 'Southlake'),
 ('United States of America', 'US', 'US', '2011 Interiors Blvd', 'South San Francisco'),
 ('United States of America', 'US', 'US', '2004 Charade Rd', 'Seattle')]
```

`LEFT JOIN` is useful when we want to retrieve data from a primary table and include related information from a secondary table, but we don't want to exclude records from the primary table if there are no matching rows in the secondary table.

And, because non-matching rows in the `locations` table are filled with the `None` values, we can apply the `LEFT JOIN` clause to find mismatched rows between the tables.

For example, to find the country that does not have any locations in the `locations` table, we use the following query.

[42]:
```
%%sql
SELECT
    country_name
FROM
    countries
LEFT JOIN locations ON locations.country_id = countries.country_id
```

(continues on next page)

(continued from previous page)

```
WHERE locations.location_id IS NULL
ORDER BY country_name;
```

```
 * sqlite://
Done.
```

[42]:
```
[('Argentina',),
 ('Australia',),
 ('Belgium',),
 ('Brazil',),
 ('China',),
 ('Denmark',),
 ('Egypt',),
 ('France',),
 ('HongKong',),
 ('India',),
 ('Israel',),
 ('Italy',),
 ('Japan',),
 ('Kuwait',),
 ('Mexico',),
 ('Netherlands',),
 ('Nigeria',),
 ('Singapore',),
 ('Switzerland',),
 ('Zambia',),
 ('Zimbabwe',)]
```

Compare the previous cell to the next cell.

[43]:
```
%%sql
SELECT
    countries.country_name,
    countries.country_id,
    locations.country_id,
    locations.street_address,
    locations.city
FROM
    countries
LEFT JOIN locations ON locations.country_id = countries.country_id
```

```
 * sqlite://
Done.
```

[43]:
```
[('Argentina', 'AR', None, None, None),
 ('Australia', 'AU', None, None, None),
 ('Belgium', 'BE', None, None, None),
 ('Brazil', 'BR', None, None, None),
 ('Canada', 'CA', 'CA', '147 Spadina Ave', 'Toronto'),
 ('Switzerland', 'CH', None, None, None),
 ('China', 'CN', None, None, None),
 ('Germany', 'DE', 'DE', 'Schwanthalerstr. 7031', 'Munich'),
 ('Denmark', 'DK', None, None, None),
 ('Egypt', 'EG', None, None, None),
```

(continues on next page)

(continued from previous page)

```
 ('France', 'FR', None, None, None),
 ('HongKong', 'HK', None, None, None),
 ('Israel', 'IL', None, None, None),
 ('India', 'IN', None, None, None),
 ('Italy', 'IT', None, None, None),
 ('Japan', 'JP', None, None, None),
 ('Kuwait', 'KW', None, None, None),
 ('Mexico', 'MX', None, None, None),
 ('Nigeria', 'NG', None, None, None),
 ('Netherlands', 'NL', None, None, None),
 ('Singapore', 'SG', None, None, None),
 ('United Kingdom', 'UK', 'UK', '8204 Arthur St', 'London'),
 ('United Kingdom', 'UK', 'UK', 'Magdalen Centre, The Oxford Science Park', 'Oxford'),
 ('United States of America', 'US', 'US', '2004 Charade Rd', 'Seattle'),
 ('United States of America', 'US', 'US', '2011 Interiors Blvd', 'South San Francisco'),
 ('United States of America', 'US', 'US', '2014 Jabberwocky Rd', 'Southlake'),
 ('Zambia', 'ZM', None, None, None),
 ('Zimbabwe', 'ZW', None, None, None)]
```

RIGHT JOIN

By analogy, `RIGHT JOIN` (or `RIGHT OUTER JOIN`) returns all rows from the right table and the matched rows from the left table. If there are no matches in the left table, the result will contain None values for those columns from the left table.

[44]:
```
%%sql
SELECT
    countries.country_name,
    countries.country_id,
    locations.country_id,
    locations.street_address,
    locations.city
FROM
    countries
RIGHT JOIN locations ON locations.country_id = countries.country_id
WHERE countries.country_id IN ('US', 'UK', 'CN', 'CA', 'DE', 'IT')
```

```
 * sqlite://
Done.
```

[44]:
```
[('Canada', 'CA', 'CA', '147 Spadina Ave', 'Toronto'),
 ('Germany', 'DE', 'DE', 'Schwanthalerstr. 7031', 'Munich'),
 ('United Kingdom', 'UK', 'UK', '8204 Arthur St', 'London'),
 ('United Kingdom', 'UK', 'UK', 'Magdalen Centre, The Oxford Science Park', 'Oxford'),
 ('United States of America', 'US', 'US', '2014 Jabberwocky Rd', 'Southlake'),
 ('United States of America', 'US', 'US', '2011 Interiors Blvd', 'South San Francisco'),
 ('United States of America', 'US', 'US', '2004 Charade Rd', 'Seattle')]
```

`FULL JOIN` (also known as a `FULL OUTER JOIN`) combines the result sets of both a `LEFT JOIN` and a `RIGHT JOIN`, it returns all rows from both tables and includes rows from the left table that have no match in the right table (with None values for right table columns) and rows from the right table that have no match in the left table (with None values for left table columns).

`CROSS JOIN` (also known as a `CARTESIAN JOIN`), combines all rows from the first table with all rows from the second table, and generates all possible combinations of data.

7.10.10 10.10 Return Data Statistics

Aggregate functions in SQL take a list of values and return a single value, such as the average `AVG()`, `MIN()`, `MAX()`, `SUM()`, or `COUNT()`.

[45]:
```
%%sql
SELECT
    AVG(salary)
FROM
    employees;
```

```
 * sqlite://
Done.
```

[45]:
```
[(8060.0,)]
```

[46]:
```
%%sql
SELECT
    MAX(salary)
FROM
    employees;
```

```
 * sqlite://
Done.
```

[46]:
```
[(24000.0,)]
```

[47]:
```
%%sql
SELECT
    SUM(salary)
FROM
    employees
WHERE department_id = 5;
```

```
 * sqlite://
Done.
```

[47]:
```
[(41200.0,)]
```

`COUNT` returns the number of rows in a table. It can be used by providing the name of a column, or it can also be used with an asterisk * as in the following cell.

[48]:
```
%%sql
SELECT
    COUNT(employee_id)
FROM
    employees;
```

```
 * sqlite://
Done.
```

[48]:
```
[(40,)]
```

[49]:
```
%%sql
SELECT
    COUNT(*)
FROM
    employees;
```

```
 * sqlite://
Done.
```

[49]:
```
[(40,)]
```

7.10.11 10.11 Grouping Data

`GROUP BY` allows to group rows based on values from more than one column. It is typically combined with aggregate functions, like `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`.

For instance, let's assume that we would like to retrieve information about the average salary in each department. In the next cell we will first display the salaries of all employees per department, and in the following cell we can see how `GROUP BY` is used to calculate the average salary for each department.

[50]:
```
%%sql
SELECT
    department_id, salary
FROM
    employees
```

```
 * sqlite://
Done.
```

[50]:
```
[(9, 24000.0),
 (9, 17000.0),
 (9, 17000.0),
 (6, 9000.0),
 (6, 6000.0),
 (6, 4800.0),
 (6, 4800.0),
 (6, 4200.0),
 (10, 12000.0),
 (10, 9000.0),
 (10, 8200.0),
 (10, 7700.0),
 (10, 7800.0),
 (10, 6900.0),
 (3, 11000.0),
 (3, 3100.0),
 (3, 2900.0),
 (3, 2800.0),
 (3, 2600.0),
 (3, 2500.0),
 (5, 8000.0),
 (5, 8200.0),
 (5, 7900.0),
 (5, 6500.0),
 (5, 2700.0),
```

(continues on next page)

(continued from previous page)

```
 (8, 14000.0),
 (8, 13500.0),
 (8, 8600.0),
 (8, 8400.0),
 (8, 7000.0),
 (8, 6200.0),
 (5, 4000.0),
 (5, 3900.0),
 (1, 4400.0),
 (2, 13000.0),
 (2, 6000.0),
 (4, 6500.0),
 (7, 10000.0),
 (11, 12000.0),
 (11, 8300.0)]
```

[51]:
```
%%sql
SELECT
    department_id, AVG(salary)
FROM
    employees
GROUP BY department_id;
```

```
 * sqlite://
Done.
```

[51]:
```
[(1, 4400.0),
 (2, 9500.0),
 (3, 4150.0),
 (4, 6500.0),
 (5, 5885.714285714285),
 (6, 5760.0),
 (7, 10000.0),
 (8, 9616.666666666666),
 (9, 19333.333333333332),
 (10, 8600.0),
 (11, 10150.0)]
```

Similarly, the following cell displays the minimum, maximum, and average salary for each department. And, instead of using the `department_id` column, it will be more convenient to display the department names. Since the `employees` table does not have a column with the department names, we will use `INNER JOIN` to retrieve the `department_name` column from the `departments` table.

[52]:
```
%%sql
SELECT
    department_name, MIN(salary) AS min_salary, MAX(salary) AS max_salary,
    ROUND(AVG(salary)) AS average_salary
FROM
    employees
INNER JOIN departments ON departments.department_id = employees.department_id
GROUP BY department_name;
```

```
 * sqlite://
Done.
```

[52]:
```
[('Accounting', 8300.0, 12000.0, 10150.0),
 ('Administration', 4400.0, 4400.0, 4400.0),
 ('Executive', 17000.0, 24000.0, 19333.0),
 ('Finance', 6900.0, 12000.0, 8600.0),
 ('Human Resources', 6500.0, 6500.0, 6500.0),
 ('IT', 4200.0, 9000.0, 5760.0),
 ('Marketing', 6000.0, 13000.0, 9500.0),
 ('Public Relations', 10000.0, 10000.0, 10000.0),
 ('Purchasing', 2500.0, 11000.0, 4150.0),
 ('Sales', 6200.0, 14000.0, 9617.0),
 ('Shipping', 2700.0, 8200.0, 5886.0)]
```

7.10.12 10.12 Modifying Data

INSERT

INSERT is used to insert one or more rows into a table, and we already used it in subsection 10.3 when we created new tables. The general syntax is:

```
INSERT INTO table (column1, column2,...)
VALUES (value1, value2, ...);
```

It is important to ensure that the number of values matches the number of columns, and that the value type corresponds to the data type for that column.

It is also possible to insert value without specifying the columns, as in:

```
INSERT INTO table
VALUES (value1, value2, ...);
```

For instance, to insert a new row into the dependents table, we can use the following code. The new row will be added to the bottom of the table dependents.

[53]:
```
%%sql
INSERT INTO
    dependents (first_name, last_name, relationship, employee_id)
VALUES ('Dustin', 'Johnson', 'Child', 178);
```

```
 * sqlite://
1 rows affected.
```

[53]:
```
[]
```

[54]:
```
# verify the changes
%sql SELECT * from dependents
```

```
 * sqlite://
Done.
```

[54]:
```
[(1, 'Penelope', 'Gietz', 'Child', 206),
 (2, 'Nick', 'Higgins', 'Child', 205),
 (3, 'Ed', 'Whalen', 'Child', 200),
 (4, 'Jennifer', 'King', 'Child', 100),
 (5, 'Johnny', 'Kochhar', 'Child', 101),
```

(continues on next page)

(continued from previous page)

```
(6, 'Bette', 'De Haan', 'Child', 102),
(7, 'Grace', 'Faviet', 'Child', 109),
(8, 'Matthew', 'Chen', 'Child', 110),
(9, 'Joe', 'Sciarra', 'Child', 111),
(10, 'Christian', 'Urman', 'Child', 112),
(11, 'Zero', 'Popp', 'Child', 113),
(12, 'Karl', 'Greenberg', 'Child', 108),
(13, 'Uma', 'Mavris', 'Child', 203),
(14, 'Vivien', 'Hunold', 'Child', 103),
(15, 'Cuba', 'Ernst', 'Child', 104),
(16, 'Fred', 'Austin', 'Child', 105),
(17, 'Helen', 'Pataballa', 'Child', 106),
(18, 'Dan', 'Lorentz', 'Child', 107),
(19, 'Bob', 'Hartstein', 'Child', 201),
(20, 'Lucille', 'Fay', 'Child', 202),
(21, 'Kirsten', 'Baer', 'Child', 204),
(22, 'Elvis', 'Khoo', 'Child', 115),
(23, 'Sandra', 'Baida', 'Child', 116),
(24, 'Cameron', 'Tobias', 'Child', 117),
(25, 'Kevin', 'Himuro', 'Child', 118),
(26, 'Rip', 'Colmenares', 'Child', 119),
(27, 'Julia', 'Raphaely', 'Child', 114),
(28, 'Woody', 'Russell', 'Child', 145),
(29, 'Alec', 'Partners', 'Child', 146),
(30, 'Sandra', 'Taylor', 'Child', 176),
(31, 'Dustin', 'Johnson', 'Child', 178)]
```

UPDATE

`UPDATE` is used to update existing data in a table.

For instance, to update the last name of the employee with `id=192`, we can write:

[55]:
```
%%sql
UPDATE employees
SET last_name = 'Lopez'
WHERE employee_id = 192;
```

```
 * sqlite://
1 rows affected.
```

[55]:
```
[]
```

And we can verify the changes.

[56]:
```
%%sql
SELECT
    employee_id, first_name, last_name
FROM
    employees
WHERE employee_id = 192;
```

```
 * sqlite://
Done.
```

[56]:
```
[(192, 'Sarah', 'Lopez')]
```

DELETE

DELETE is used to delete one or more rows from a table permanently.

[57]:
```
%%sql
DELETE FROM dependents
WHERE dependent_id = 16;
```

```
 * sqlite://
1 rows affected.
```

[57]:
```
[]
```

[58]:
```
# verify the changes

%sql SELECT * FROM dependents
```

```
 * sqlite://
Done.
```

[58]:
```
[(1, 'Penelope', 'Gietz', 'Child', 206),
 (2, 'Nick', 'Higgins', 'Child', 205),
 (3, 'Ed', 'Whalen', 'Child', 200),
 (4, 'Jennifer', 'King', 'Child', 100),
 (5, 'Johnny', 'Kochhar', 'Child', 101),
 (6, 'Bette', 'De Haan', 'Child', 102),
 (7, 'Grace', 'Faviet', 'Child', 109),
 (8, 'Matthew', 'Chen', 'Child', 110),
 (9, 'Joe', 'Sciarra', 'Child', 111),
 (10, 'Christian', 'Urman', 'Child', 112),
 (11, 'Zero', 'Popp', 'Child', 113),
 (12, 'Karl', 'Greenberg', 'Child', 108),
 (13, 'Uma', 'Mavris', 'Child', 203),
 (14, 'Vivien', 'Hunold', 'Child', 103),
 (15, 'Cuba', 'Ernst', 'Child', 104),
 (17, 'Helen', 'Pataballa', 'Child', 106),
 (18, 'Dan', 'Lorentz', 'Child', 107),
 (19, 'Bob', 'Hartstein', 'Child', 201),
 (20, 'Lucille', 'Fay', 'Child', 202),
 (21, 'Kirsten', 'Baer', 'Child', 204),
 (22, 'Elvis', 'Khoo', 'Child', 115),
 (23, 'Sandra', 'Baida', 'Child', 116),
 (24, 'Cameron', 'Tobias', 'Child', 117),
 (25, 'Kevin', 'Himuro', 'Child', 118),
 (26, 'Rip', 'Colmenares', 'Child', 119),
 (27, 'Julia', 'Raphaely', 'Child', 114),
 (28, 'Woody', 'Russell', 'Child', 145),
 (29, 'Alec', 'Partners', 'Child', 146),
 (30, 'Sandra', 'Taylor', 'Child', 176),
 (31, 'Dustin', 'Johnson', 'Child', 178)]
```

7.10.13 10.13 Working with Tables

In subsection 10.3 we learned how to use the `CREATE TABLE` statement. Several other related SQL statements for working with tables are described next.

ALTER TABLE

`ALTER TABLE` allows to add new columns in an existing table using `ADD`, remove columns in a table with `DROP`, rename columns with `RENAME`, or modify attributes of a column such as primary key, default value, etc. with `ADD CONSTRAINT` or `DROP CONSTRAINT` and related commands.

Add a Column

[59]:
```
%%sql
ALTER TABLE cars
ADD mileage INT;
```

```
 * sqlite://
Done.
```

[59]:
```
[]
```

[60]:
```
%sql SELECT * FROM cars
```

```
 * sqlite://
Done.
```

[60]:
```
[(1, 'Audi', 52642, None),
 (2, 'Mercedes', 57127, None),
 (3, 'Skoda', 9000, None),
 (4, 'Volvo', 29000, None),
 (5, 'Bentley', 350000, None),
 (6, 'Citroen', 21000, None),
 (7, 'Hummer', 41400, None),
 (8, 'Volkswagen', 21600, None)]
```

Remove a Column

[61]:
```
%%sql
ALTER TABLE cars
DROP price;
```

```
 * sqlite://
Done.
```

[61]:
```
[]
```

[62]:
```
# verfiy the change
%sql SELECT * FROM cars
```

```
 * sqlite://
Done.
```

[62]:
```
[(1, 'Audi', None),
 (2, 'Mercedes', None),
 (3, 'Skoda', None),
 (4, 'Volvo', None),
```

(continues on next page)

(continued from previous page)

```
 (5, 'Bentley', None),
 (6, 'Citroen', None),
 (7, 'Hummer', None),
 (8, 'Volkswagen', None)]
```

Rename a Column

[63]:
```
%%sql
ALTER TABLE cars
RENAME mileage TO miles;
```

```
 * sqlite://
Done.
```

[63]:
```
[]
```

[64]:
```
# verfiy the change
%sql SELECT * FROM cars
```

```
 * sqlite://
Done.
```

[64]:
```
[(1, 'Audi', None),
 (2, 'Mercedes', None),
 (3, 'Skoda', None),
 (4, 'Volvo', None),
 (5, 'Bentley', None),
 (6, 'Citroen', None),
 (7, 'Hummer', None),
 (8, 'Volkswagen', None)]
```

Rename a Table

Similarly, `ALTER TABLE` can be used to rename a table, as in the following example.

[65]:
```
%%sql
ALTER TABLE writer
RENAME TO authors;
```

```
 * sqlite://
Done.
```

[65]:
```
[]
```

[66]:
```
# verfiy the change
%sql SELECT name FROM sqlite_master WHERE type='table'
```

```
 * sqlite://
Done.
```

[66]:
```
[('cars',),
 ('authors',),
 ('regions',),
 ('sqlite_sequence',),
 ('countries',),
 ('locations',),
```

(continues on next page)

(continued from previous page)

```
 ('departments',),
 ('jobs',),
 ('employees',),
 ('dependents',)]
```

DROP TABLE

`DROP TABLE` removes a table from a database.

[67]:
```
%sql DROP TABLE cars;
```

```
 * sqlite://
Done.
```

[67]:
```
[]
```

[68]:
```
# verfiy the change
%sql SELECT name FROM sqlite_master WHERE type='table'
```

```
 * sqlite://
Done.
```

[68]:
```
[('authors',),
 ('regions',),
 ('sqlite_sequence',),
 ('countries',),
 ('locations',),
 ('departments',),
 ('jobs',),
 ('employees',),
 ('dependents',)]
```

7.10.14 10.14 Constraints

Constraints enforce restrictions on the data that a column can contain or impose other related restrictions. So far, we described some of the constraints used in SQL databases, such as:

- `PRIMARY KEY`, defines a primary key for a table.
- `NOT NULL`, ensures that values are inserted in a column.
- `UNIQUE`, ensures that each row in a column has unique values.

FOREIGN KEY

Beside the primary key, some tables also define a **foreign key**. This is a column or a group of columns that enforces a link between the data in two tables. In a foreign key reference, the primary key column of the first table is referenced by the column of the second table, and the column of the second table becomes the foreign key.

For instance, when we created the `countries` table (see the inserted code below), we specified that it has `country_id` column as a primary key, and `region_id` column as a foreign key. Therefore, the `region_id` column in `countries` will point to the `region_id` column in the `regions` table. This is indicated in the last line below: `FOREIGN KEY (region_id) REFERENCES regions (region_id)`.

```
CREATE TABLE countries (
    country_id TEXT NOT NULL,
    country_name TEXT NOT NULL,
    region_id INTEGER NOT NULL,
    PRIMARY KEY (country_id ASC),
    FOREIGN KEY (region_id) REFERENCES regions (region_id) ON DELETE CASCADE ON UPDATE␣
↪CASCADE);
```

Note also that in the last line `ON DELETE CASCADE` specifies that if a row in the `regions` table is deleted, all rows in the `countries` table that have a matching `region_id` will also be deleted automatically. Analogously, `ON UPDATE CASCADE` specifies that if a `region_id` value in the `regions` table is updated, all corresponding rows in the `countries` table will be updated accordingly. This is useful for maintaining referential integrity between the rows in these two tables.

Let's inspect the `countries` table.

[69]:
```
%sql SELECT * FROM countries
```

```
 * sqlite://
Done.
```

[69]:
```
[('AR', 'Argentina', 2),
 ('AU', 'Australia', 3),
 ('BE', 'Belgium', 1),
 ('BR', 'Brazil', 2),
 ('CA', 'Canada', 2),
 ('CH', 'Switzerland', 1),
 ('CN', 'China', 3),
 ('DE', 'Germany', 1),
 ('DK', 'Denmark', 1),
 ('EG', 'Egypt', 4),
 ('FR', 'France', 1),
 ('HK', 'HongKong', 3),
 ('IL', 'Israel', 4),
 ('IN', 'India', 3),
 ('IT', 'Italy', 1),
 ('JP', 'Japan', 3),
 ('KW', 'Kuwait', 4),
 ('MX', 'Mexico', 2),
 ('NG', 'Nigeria', 4),
 ('NL', 'Netherlands', 1),
 ('SG', 'Singapore', 3),
 ('UK', 'United Kingdom', 1),
 ('US', 'United States of America', 2),
 ('ZM', 'Zambia', 4),
 ('ZW', 'Zimbabwe', 4)]
```

[70]:
```
%sql SELECT * FROM regions
```

```
 * sqlite://
Done.
```

[70]:
```
[(1, 'Europe'), (2, 'Americas'), (3, 'Asia'), (4, 'Middle East and Africa')]
```

As we mentioned in the section on joining tables, we can use the foreign key column in a table to link to other tables. In the above example, the `region_id` column establishes a relationship between the `countries` table and the `regions`

table, where`countries.region_id` must match an existing value in `regions.region_id`.

7.10.15 10.15 Subqueries

A *subquery* is a query that is nested inside another query, such as `SELECT`.

For instance, the following cell retrieves employees in the departments that have a `location_id=1700`.

The query placed inside the parentheses is a subquery. It is also known as an *inner query* or *inner select*. The query that contains the subquery is called an *outer query* or an *outer select*.

To execute the query, the database system first has to execute the subquery and retrieve the departments with a `location_id` of 1700, and afterward it has to execute the outer query.

[71]:
```
%%sql
SELECT
    employee_id, first_name, last_name
FROM
    employees
WHERE department_id IN
    (SELECT department_id FROM departments
        WHERE location_id = 1700)
ORDER BY first_name , last_name;
```

```
 * sqlite://
Done.
```

[71]:
```
[(115, 'Alexander', 'Khoo'),
 (109, 'Daniel', 'Faviet'),
 (114, 'Den', 'Raphaely'),
 (118, 'Guy', 'Himuro'),
 (111, 'Ismael', 'Sciarra'),
 (200, 'Jennifer', 'Whalen'),
 (110, 'John', 'Chen'),
 (112, 'Jose Manuel', 'Urman'),
 (119, 'Karen', 'Colmenares'),
 (102, 'Lex', 'De Haan'),
 (113, 'Luis', 'Popp'),
 (108, 'Nancy', 'Greenberg'),
 (101, 'Neena', 'Kochhar'),
 (205, 'Shelley', 'Higgins'),
 (116, 'Shelli', 'Baida'),
 (117, 'Sigal', 'Tobias'),
 (100, 'Steven', 'King'),
 (206, 'William', 'Gietz')]
```

The next code finds the employee who has the highest salary.

[72]:
```
%%sql
SELECT
    employee_id, first_name, last_name, salary
FROM
    employees
WHERE salary =
    (SELECT MAX(salary) FROM employees)
ORDER BY first_name , last_name;
```

```
 * sqlite://
Done.
```

[72]:
```
[(100, 'Steven', 'King', 24000.0)]
```

E.g., find all employees who salaries are greater than the average salary of all employees.

[73]:
```
%%sql
SELECT
    employee_id, first_name, last_name, salary
FROM
    employees
WHERE
    salary >
    (SELECT AVG(salary) FROM employees);
```

```
 * sqlite://
Done.
```

[73]:
```
[(100, 'Steven', 'King', 24000.0),
 (101, 'Neena', 'Kochhar', 17000.0),
 (102, 'Lex', 'De Haan', 17000.0),
 (103, 'Alexander', 'Hunold', 9000.0),
 (108, 'Nancy', 'Greenberg', 12000.0),
 (109, 'Daniel', 'Faviet', 9000.0),
 (110, 'John', 'Chen', 8200.0),
 (114, 'Den', 'Raphaely', 11000.0),
 (121, 'Adam', 'Fripp', 8200.0),
 (145, 'John', 'Russell', 14000.0),
 (146, 'Karen', 'Partners', 13500.0),
 (176, 'Jonathon', 'Taylor', 8600.0),
 (177, 'Jack', 'Livingston', 8400.0),
 (201, 'Michael', 'Hartstein', 13000.0),
 (204, 'Hermann', 'Baer', 10000.0),
 (205, 'Shelley', 'Higgins', 12000.0),
 (206, 'William', 'Gietz', 8300.0)]
```

7.10.16 10.16 Connect to an Existing Database

Recall again that when we created new tables in section 10.3 or a new database in section 10.4, we used `sqlite://` to connect to the tables or to the database. This syntax is used to specify an **in-memory** table or database, that is, it indicates that we want to work with tables or databases that exist only in the memory of our local machine. This syntax uses two forward slashes `//` after `sqlite:`.

To connect to an existing database that is stored in a local directory of our computer, we will use the syntax `sqlite:///relative_database_path`, where `relative_database_path` specifies the **relative path** to the local directory where the SQL database is stored. This syntax uses three forward slashes `///` after `sqlite:`.

Similarly, we can connect to an existing database that is stored in a directory of our computer by using the **absolute path** to the database with the syntax `sqlite:////absolute_database_path`. This syntax uses four forward slashes `////` after `sqlite:`.

In the following cell, we used `sqlite:///data/EssentialSQL.db` to initialize the connection to the database `EssentialSQL.db` which is located in the `data` subdirectory in the current working directory.

[74]:
```
%sql sqlite:///data/EssentialSQL.db
```

Note again that there are three forward slashes after `sqlite:`, since we provided a relative path to the database.

We can inspect the tables in the database `EssentialSQL.db` in the following cell.

[75]:
```
%sql SELECT name FROM sqlite_master WHERE type='table'
```

```
   sqlite://
 * sqlite:///data/EssentialSQL.db
Done.
```

[75]:
```
[('Customers',), ('Shippers',), ('Employees',), ('Orders',), ('OrderDetails',)]
```

And we can inspect the data in the table `Customers`.

[76]:
```
%sql SELECT * FROM Customers
```

```
   sqlite://
 * sqlite:///data/EssentialSQL.db
Done.
```

[76]:
```
[(1, 'Deerfield Tile', 'Dick Terrcotta', 'Owner', '450 Village Street', 'Deerfield', 'IL
↪'),
 (2, 'Sagebrush Carpet', 'Barbara Berber', 'Director of Installations', '10 Industrial␣
↪Drive', 'El Paso', 'TX'),
 (3, 'Floor Co.', 'Jim Wood', 'Installer', '34218 Private Lane', 'Monclair', 'NJ'),
 (4, 'Main Tile and Bath', 'Toni Faucet', 'Owner', 'Suite 23, Henry Building', 'Orlando',
↪ 'FL'),
 (5, 'Slots Carpet', 'Jack Diamond III', 'Purchaser', '3024 Jackpot Drive', 'Las Vegas',
↪'NV')]
```

If the database does not exist, SQLite will create a new database with the provided name in the home directory. The following cell will create a new database named `test.db` in the current working directory.

[77]:
```
%sql sqlite:///test.db
```

7.10.17 References

1. SQL Tutorial, available at https://www.sqltutorial.org/.
2. Practice SQL with SQLite and Jupyter Notebook, by Chonghua Yin, available at https://github.com/royalosyin/Practice-SQL-with-SQLite-and-Jupyter-Notebook.

BACK TO TOP

7.11 Lecture 11 - Data Exploration and Preprocessing

7.11.1 11.1 Exploratory Data Analysis

Exploratory Data Analysis (EDA) is an important step in all data science projects, and involves several exploratory steps to obtain a better understanding of the data.

EDA typically includes: inspecting the summary statistics of the data, observing if there are missing values and adopting an appropriate strategy for handling them, checking the distribution of the features and whether there is a correlation between features, understanding which features are important and worth keeping and which ones are less important, and similar.

To provide an example of EDA, we will use the `Titanic` dataset which can be loaded from the `Seaborn` datasets.

[1]:
```
# import libraries
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

[2]:
```
titanic = sns.load_dataset('titanic')
```

Let's check the basic information about the data. There are 891 rows (samples) and 15 columns (features). We can see below the data types of each column.

[3]:
```
titanic.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
```

(continues on next page)

(continued from previous page)

```
 0   survived     891 non-null    int64
 1   pclass       891 non-null    int64
 2   sex          891 non-null    object
 3   age          714 non-null    float64
 4   sibsp        891 non-null    int64
 5   parch        891 non-null    int64
 6   fare         891 non-null    float64
 7   embarked     889 non-null    object
 8   class        891 non-null    category
 9   who          891 non-null    object
 10  adult_male   891 non-null    bool
 11  deck         203 non-null    category
 12  embark_town  889 non-null    object
 13  alive        891 non-null    object
 14  alone        891 non-null    bool
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
memory usage: 80.7+ KB
```

Let's display the first five rows and the last five rows. As we can notice, each row presents data for one passenger on Titanic.

[4]:
```
titanic.head()
```

[4]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

[5]:
```
titanic.tail()
```

[5]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
886	0	2	male	27.0	0	0	13.00	S	Second	man	True	NaN	Southampton	no	True
887	1	1	female	19.0	0	0	30.00	S	First	woman	False	B	Southampton	yes	True
888	0	3	female	NaN	1	2	23.45	S	Third	woman	False	NaN	Southampton	no	False
889	1	1	male	26.0	0	0	30.00	C	First	man	True	C	Cherbourg	yes	True
890	0	3	male	32.0	0	0	7.75	Q	Third	man	True	NaN	Queenstown	no	True

Let's also see the summary statistic. Recall that statistics are shown only for the columns with numerical data.

[6]:
```
titanic.describe()
```

[6]:
```
         survived      pclass         age       sibsp       parch        fare
count  891.000000  891.000000  714.000000  891.000000  891.000000  891.000000
mean     0.383838    2.308642   29.699118    0.523008    0.381594   32.204208
std      0.486592    0.836071   14.526497    1.102743    0.806057   49.693429
min      0.000000    1.000000    0.420000    0.000000    0.000000    0.000000
25%      0.000000    2.000000   20.125000    0.000000    0.000000    7.910400
50%      0.000000    3.000000   28.000000    0.000000    0.000000   14.454200
75%      1.000000    3.000000   38.000000    1.000000    0.000000   31.000000
max      1.000000    3.000000   80.000000    8.000000    6.000000  512.329200
```

Let's assume that our task is to predict whether the passenger shown in the next cell survived. I.e., we will take the `survived` column to be the target, and we will implement a classification algorithm to predict it based on the other columns in the dataset.

[7]:
```
titanic.head(1)
```

[7]:
```
   survived  pclass   sex   age  sibsp  parch  fare embarked  class  who  \
0         0       3  male  22.0      1      0  7.25        S  Third  man

   adult_male deck  embark_town alive  alone
0        True  NaN  Southampton    no  False
```

Explore Column Information

Let's first inspect some of the columns in the dataset.

For instance, we can use the following code to find how many passengers survived and how many died.

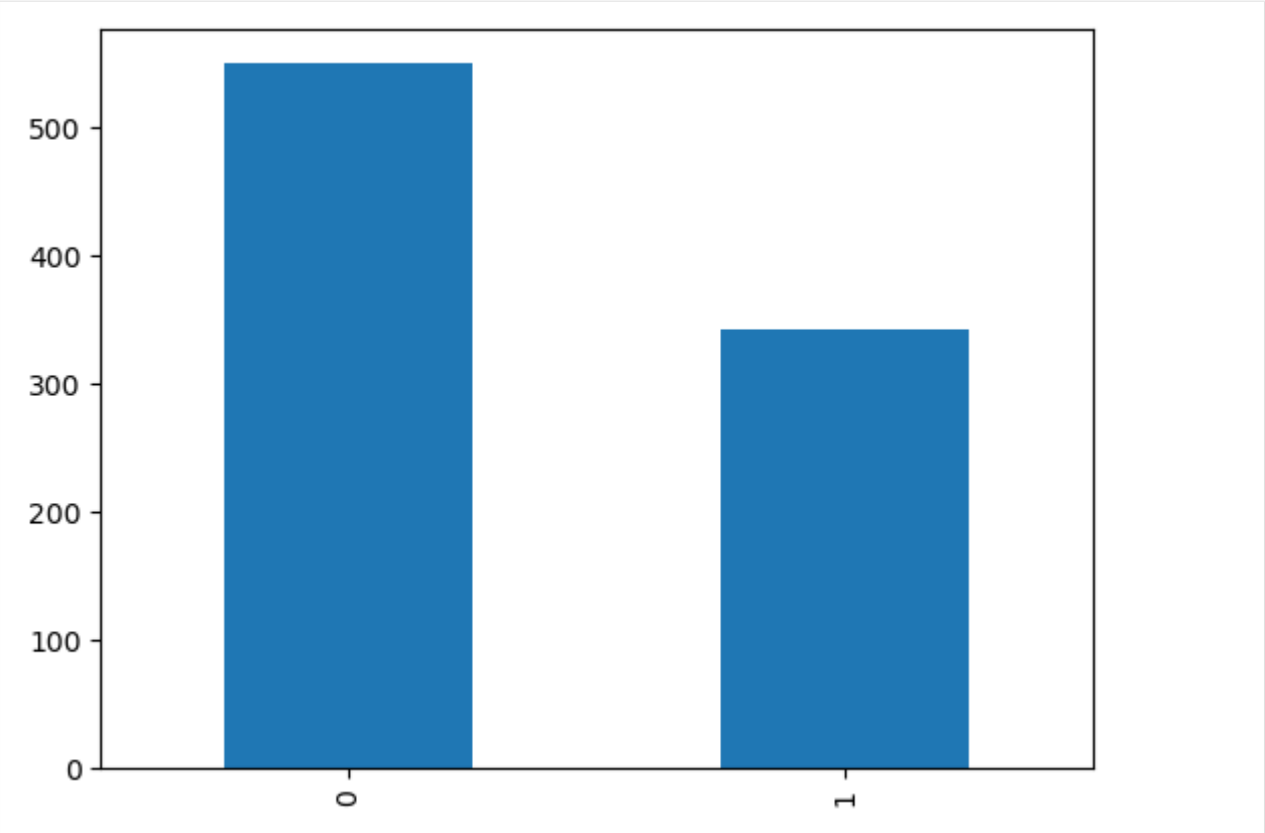
[8]:
```
titanic['survived'].value_counts()
```

[8]:
```
0    549
1    342
Name: survived, dtype: int64
```

It is often easier to understand the data if we plot the values. The pandas library provides basic plotting functions. For instance, in the next cell we created a bar plot by using `plot(kind='bar')` directly in pandas. The syntax for plotting in `pandas` is somewhat different than the `matplotlib` functions, and admittedly, the functionality for plotting in pandas is limited. We will learn later about the `Seaborn` library which allows plotting directly in DataFrames and provides improved visualizations in comparison to pandas plots.

[9]:
```
titanic['survived'].value_counts().plot(kind='bar')
```

[9]:
```
<Axes: >
```



Notice that there is a column called `alive` which is the same as the `survived` column. We need to remove this column from the data, otherwise the classifier will just use that column to make predictions for the survived passengers, and will achieve 100% accuracy.

[10]:
```
titanic['alive'].value_counts()
```

[10]:
```
no     549
yes    342
Name: alive, dtype: int64
```

[11]:
```
titanic.drop(['alive'], axis=1, inplace=True)
```

[12]:
```
# verify the change
titanic.head()
```

[12]:
```
   survived  pclass     sex   age  sibsp  parch     fare embarked  class  \
0         0       3    male  22.0      1      0   7.2500        S  Third
1         1       1  female  38.0      1      0  71.2833        C  First
2         1       3  female  26.0      0      0   7.9250        S  Third
3         1       1  female  35.0      1      0  53.1000        S  First
4         0       3    male  35.0      0      0   8.0500        S  Third

     who  adult_male deck  embark_town  alone
0    man        True  NaN  Southampton  False
1  woman       False    C    Cherbourg  False
2  woman       False  NaN  Southampton   True
3  woman       False    C  Southampton  False
```

(continues on next page)

(continued from previous page)

```
4    man        True  NaN  Southampton   True
```

Also, there are two columns called `class` and `plass`. Let's examine how many passengers are in these columns.

[13]:
```
titanic['pclass'].value_counts()
```

[13]:
```
3    491
1    216
2    184
Name: pclass, dtype: int64
```

[14]:
```
titanic['class'].value_counts()
```

[14]:
```
Third     491
First     216
Second    184
Name: class, dtype: int64
```

[15]:
```
# compare the two columns
p_class = titanic[['pclass', 'class']]
p_class.head()
```

[15]:
```
   pclass  class
0       3  Third
1       1  First
2       3  Third
3       1  First
4       3  Third
```

It seems that both of these columns are the same, except that one is numeric and the other contains text. Let's drop the `class` column.

[16]:
```
titanic.drop(['class'], axis=1, inplace=True)
```

[17]:
```
# verify the change
titanic.head()
```

[17]:
```
   survived  pclass     sex   age  sibsp  parch     fare embarked    who  \
0         0       3    male  22.0      1      0   7.2500        S    man
1         1       1  female  38.0      1      0  71.2833        C  woman
2         1       3  female  26.0      0      0   7.9250        S  woman
3         1       1  female  35.0      1      0  53.1000        S  woman
4         0       3    male  35.0      0      0   8.0500        S    man

   adult_male deck  embark_town  alone
0        True  NaN  Southampton  False
1       False    C    Cherbourg  False
2       False  NaN  Southampton   True
3       False    C  Southampton  False
4        True  NaN  Southampton   True
```

Let's explore the two columns `embarked` and `embark_town`.

[18]:
```
titanic['embarked'].value_counts()
```

[18]:
```
S    644
C    168
Q     77
Name: embarked, dtype: int64
```

[19]:
```
titanic['embark_town'].value_counts()
```

[19]:
```
Southampton    644
Cherbourg      168
Queenstown      77
Name: embark_town, dtype: int64
```

They are the same, therefore, drop `embarked`.

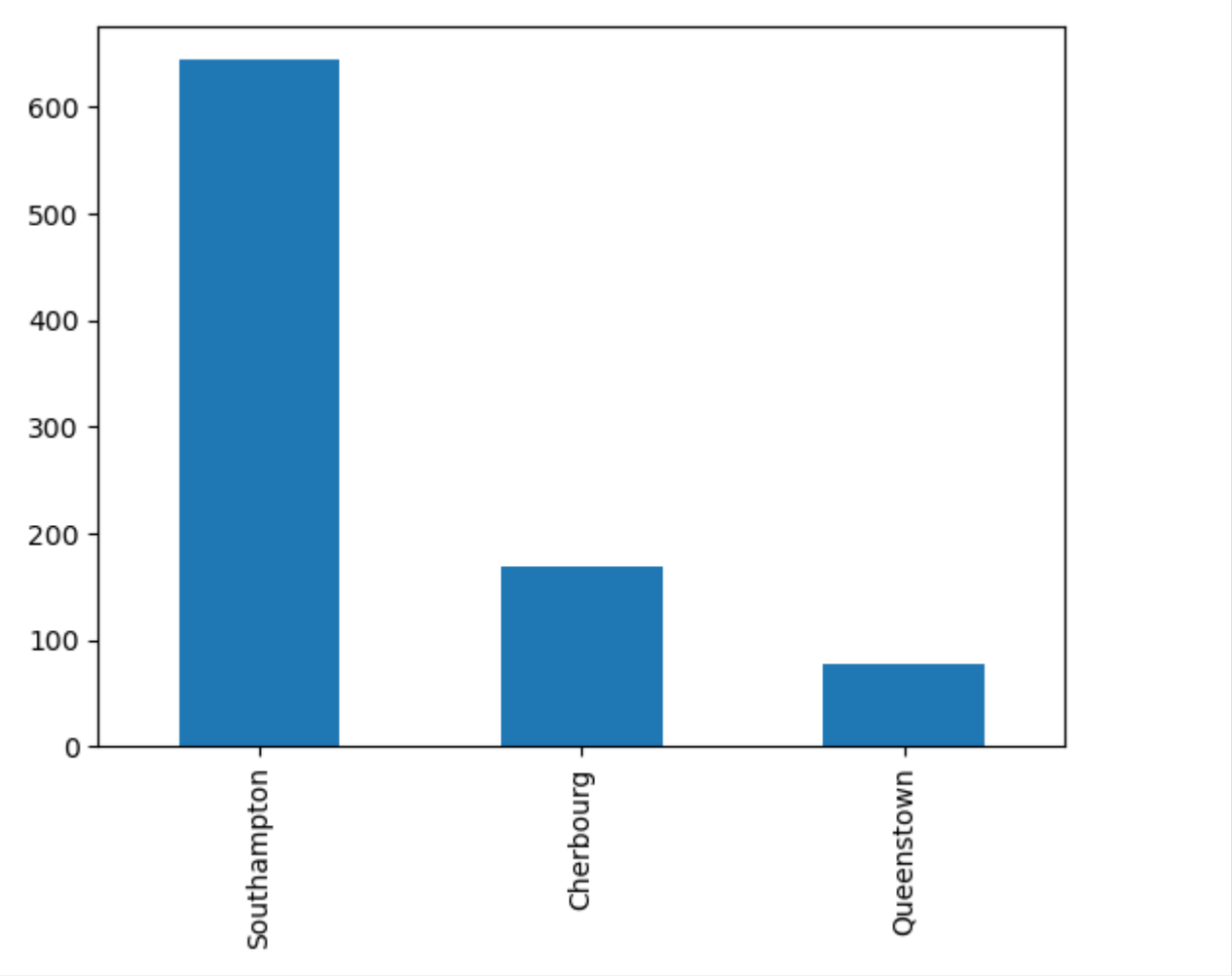
[20]:
```
titanic.drop(['embarked'], axis=1, inplace=True)
```

[21]:
```
# verify the change
titanic.head()
```

[21]:
```
   survived  pclass     sex   age  sibsp  parch     fare    who  adult_male  \
0         0       3    male  22.0      1      0   7.2500    man        True
1         1       1  female  38.0      1      0  71.2833  woman       False
2         1       3  female  26.0      0      0   7.9250  woman       False
3         1       1  female  35.0      1      0  53.1000  woman       False
4         0       3    male  35.0      0      0   8.0500    man        True

  deck  embark_town  alone
0  NaN  Southampton  False
1    C    Cherbourg  False
2  NaN  Southampton   True
3    C  Southampton  False
4  NaN  Southampton   True
```

Let's plot the occurrences in `embark_town` in a bar plot.

[22]:
```
titanic['embark_town'].value_counts().plot(kind='bar')
plt.show()
```



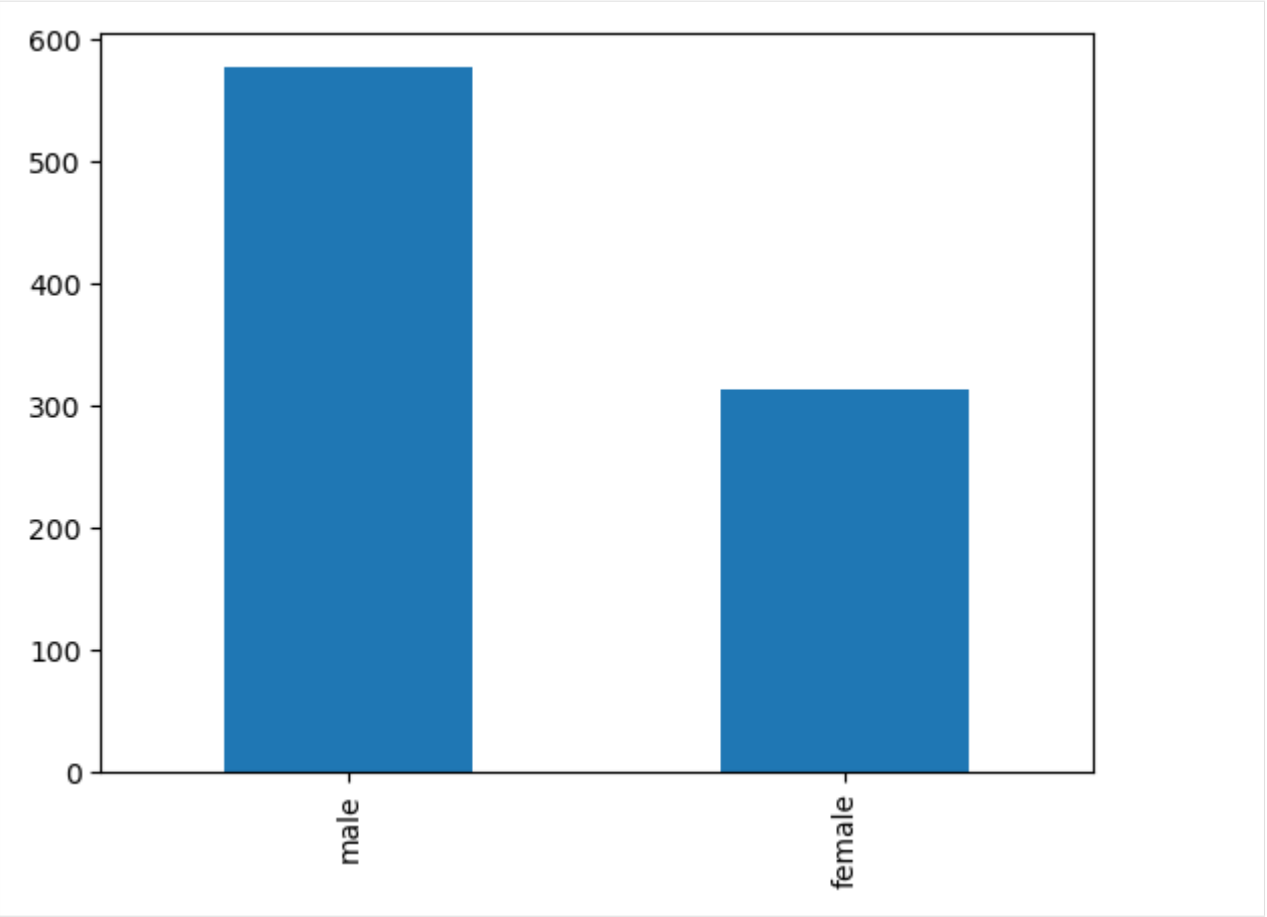
Also, let's check how many men and women there are in the dataset.

[23]:
```
titanic['sex'].value_counts()
```

[23]:
```
male      577
female    314
Name: sex, dtype: int64
```

[24]:
```
titanic['sex'].value_counts().plot(kind='bar')
```

[24]:
```
<Axes: >
```



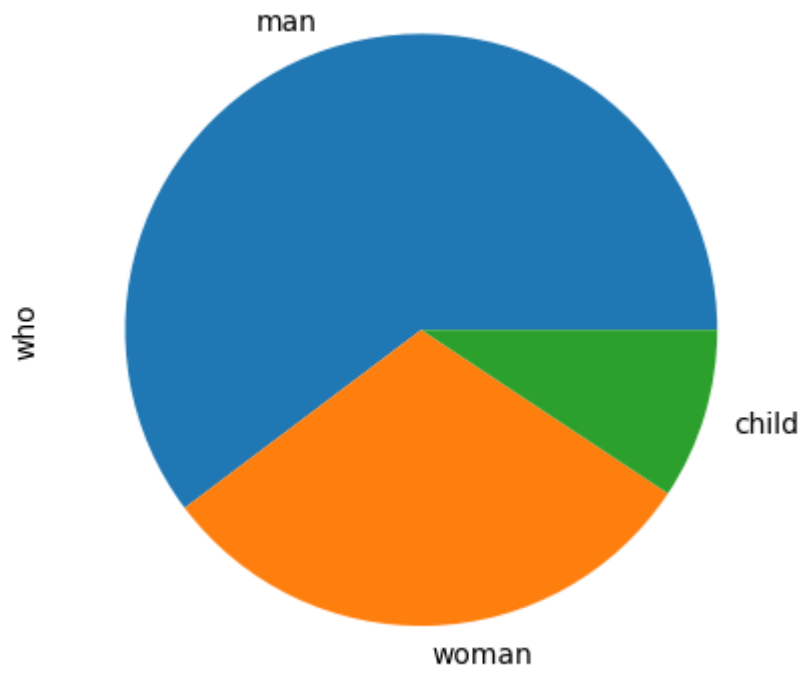
[25]:
```
titanic['who'].value_counts()
```

[25]:
```
man      537
woman    271
child     83
Name: who, dtype: int64
```

We can note that the column `who` is similar to the `sex` column, but it also includes the number of children.

As an exercise, let's show the categories of the column `who` using a Pie Chart to visualize their values.

[26]:
```
titanic.who.value_counts().plot(kind='pie')
plt.show()
```



There is another column `adult_male` which is similar, but different than `sex` and `who`.

[27]:
```
titanic['adult_male'].value_counts()
```

[27]:
```
True     537
False    354
Name: adult_male, dtype: int64
```

Missing Data

Let's check which columns have data missing.

[28]:
```
titanic.isnull().sum()
```

[28]:
```
survived         0
pclass           0
sex              0
age            177
sibsp            0
parch            0
fare             0
who              0
adult_male       0
deck           688
embark_town      2
alone            0
dtype: int64
```

There are missing data in `age`, `deck`, and `embark_town` columns.

The `deck` column has too many rows missing, and probably the deck on which the passenger boarded the ship is not too important for the task of predicting the survived passengers, thus, let's drop it.

[29]:
```
titanic.drop(['deck'], axis=1, inplace=True)
```

[30]:
```
# verify the change
titanic.head()
```

[30]:
```
   survived  pclass     sex   age  sibsp  parch     fare    who  adult_male  \
0         0       3    male  22.0      1      0   7.2500    man        True
1         1       1  female  38.0      1      0  71.2833  woman       False
2         1       3  female  26.0      0      0   7.9250  woman       False
3         1       1  female  35.0      1      0  53.1000  woman       False
4         0       3    male  35.0      0      0   8.0500    man        True

   embark_town  alone
0  Southampton  False
1    Cherbourg  False
2  Southampton   True
3  Southampton  False
4  Southampton   True
```

Since only 2 rows are missing in `embark town` let's remove those two rows. In the next cell we used `dropna` to remove only those rows that have missing values in the the `embark town` column.

Recall again that to drop columns in pandas we use `axis=1` and to drop rows we use `axis=0`.

[31]:
```
titanic.dropna(subset=['embark_town'], axis=0, inplace=True)
```

We can notice that the number of rows was reduced to 889 from the original 891, because of the removed 2 rows.

[32]:
```
# verify the change
titanic.shape
```

[32]:
```
(889, 11)
```

Now we have Nan values only in the `age` column.

[33]:
```
titanic.isnull().sum()
```

[33]:
```
survived         0
pclass           0
sex              0
age            177
sibsp            0
parch            0
fare             0
who              0
adult_male       0
embark_town      0
alone            0
dtype: int64
```

There are several ways to deal with this. One is to replace the missing values in the `age` column with the average value of the age of passengers, or with some other value (e.g., 0 in some cases).

Let's first explore the first option, and let's create a new DataFrame called `titanic_filled` which replaces the missing values in the `age` column with the average age. For this purpose we will use the method `fillna` that will calculate the mean and fill in the missing values.

[34]:
```
titanic_filled = titanic.fillna(titanic.mean(axis=0))
```

```
C:\Users\vakanski\AppData\Local\Temp\ipykernel_10716\1546117764.py:1: FutureWarning: The␣
↪default value of numeric_only in DataFrame.mean is deprecated. In a future version, it␣
↪will default to False. In addition, specifying 'numeric_only=None' is deprecated.␣
↪Select only valid columns or specify the value of numeric_only to silence this warning.
  titanic_filled = titanic.fillna(titanic.mean(axis=0))
```

To verify the above, let's display several rows that have missing values for the `age`, and let's display below the DataFrame with the filled values. We can note that the average age is `29.64` years.

[35]:
```
titanic.head(8)
```

[35]:

	survived	pclass	sex	age	sibsp	parch	fare	who	adult_male	\
0	0	3	male	22.0	1	0	7.2500	man	True	
1	1	1	female	38.0	1	0	71.2833	woman	False	
2	1	3	female	26.0	0	0	7.9250	woman	False	
3	1	1	female	35.0	1	0	53.1000	woman	False	
4	0	3	male	35.0	0	0	8.0500	man	True	
5	0	3	male	NaN	0	0	8.4583	man	True	
6	0	1	male	54.0	0	0	51.8625	man	True	
7	0	3	male	2.0	3	1	21.0750	child	False	

	embark_town	alone
0	Southampton	False
1	Cherbourg	False
2	Southampton	True
3	Southampton	False
4	Southampton	True
5	Queenstown	True
6	Southampton	True
7	Southampton	False

[36]:
```
titanic_filled.head(8)
```

[36]:

	survived	pclass	sex	age	sibsp	parch	fare	who	\
0	0	3	male	22.000000	1	0	7.2500	man	
1	1	1	female	38.000000	1	0	71.2833	woman	
2	1	3	female	26.000000	0	0	7.9250	woman	
3	1	1	female	35.000000	1	0	53.1000	woman	
4	0	3	male	35.000000	0	0	8.0500	man	
5	0	3	male	29.642093	0	0	8.4583	man	
6	0	1	male	54.000000	0	0	51.8625	man	
7	0	3	male	2.000000	3	1	21.0750	child	

	adult_male	embark_town	alone
0	True	Southampton	False
1	False	Cherbourg	False
2	False	Southampton	True
3	False	Southampton	False
4	True	Southampton	True

(continues on next page)

(continued from previous page)

```
5        True   Queenstown   True
6        True  Southampton   True
7       False  Southampton  False
```

We can observe now that there are no missing values in the `titanic_filled` DataFrame.

[37]:
```
titanic_filled.isnull().sum()
```

[37]:
```
survived       0
pclass         0
sex            0
age            0
sibsp          0
parch          0
fare           0
who            0
adult_male     0
embark_town    0
alone          0
dtype: int64
```

Another alternative is to drop the rows with the missing values for the `age`. Let's explore this strategy as well.

In general, we can try both strategies and check which one produces better results with the classification algorithm.

After dropping the rows with missing values, 712 rows are remaining in the dataset. The method `reset_index` will change the index column to range from 0 to 711. If we didn't reset the index column, the index values would have still ranged from 0 to 891.

[38]:
```
titanic.dropna(inplace=True)
```

[39]:
```
titanic.reset_index(inplace=True)
```

[40]:
```
titanic.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 712 entries, 0 to 711
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   index        712 non-null    int64
 1   survived     712 non-null    int64
 2   pclass       712 non-null    int64
 3   sex          712 non-null    object
 4   age          712 non-null    float64
 5   sibsp        712 non-null    int64
 6   parch        712 non-null    int64
 7   fare         712 non-null    float64
 8   who          712 non-null    object
 9   adult_male   712 non-null    bool
 10  embark_town  712 non-null    object
 11  alone        712 non-null    bool
dtypes: bool(2), float64(2), int64(5), object(3)
memory usage: 57.1+ KB
```

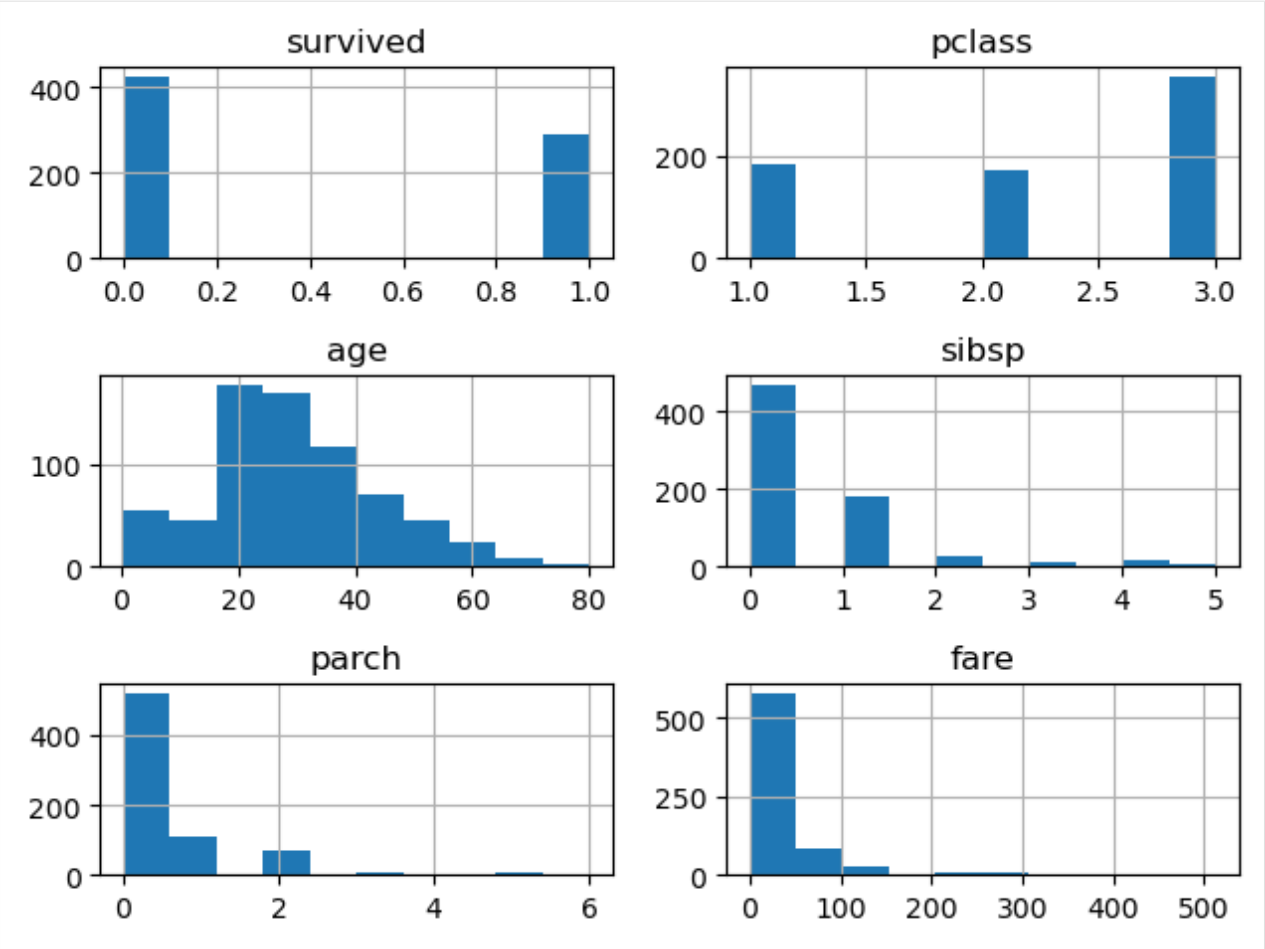
[41]:
```
titanic.head()
```

[41]:
```
   index  survived  pclass     sex   age  sibsp  parch     fare    who  \
0      0         0       3    male  22.0      1      0   7.2500    man
1      1         1       1  female  38.0      1      0  71.2833  woman
2      2         1       3  female  26.0      0      0   7.9250  woman
3      3         1       1  female  35.0      1      0  53.1000  woman
4      4         0       3    male  35.0      0      0   8.0500    man

   adult_male  embark_town  alone
0        True  Southampton  False
1       False    Cherbourg  False
2       False  Southampton   True
3       False  Southampton  False
4        True  Southampton   True
```

Checking Feature Distribution

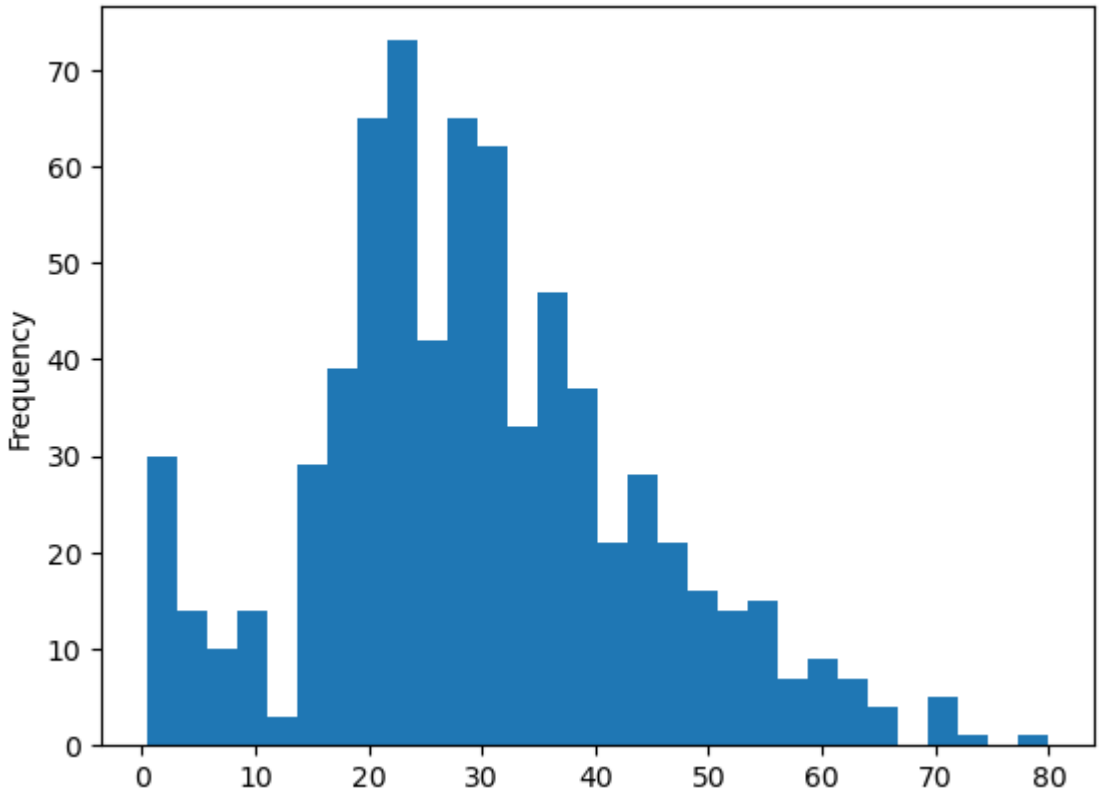
Let's check the distribution of the numerical columns in the dataset, by plotting the histograms. For the columns with categorical data, such as `survived` and `pclass`, the histograms are equivalent to bar plots, and are less helpful.

[42]:
```
titanic[['survived','pclass','age','sibsp','parch','fare']].hist(bins=10)
plt.tight_layout()
plt.show()
```



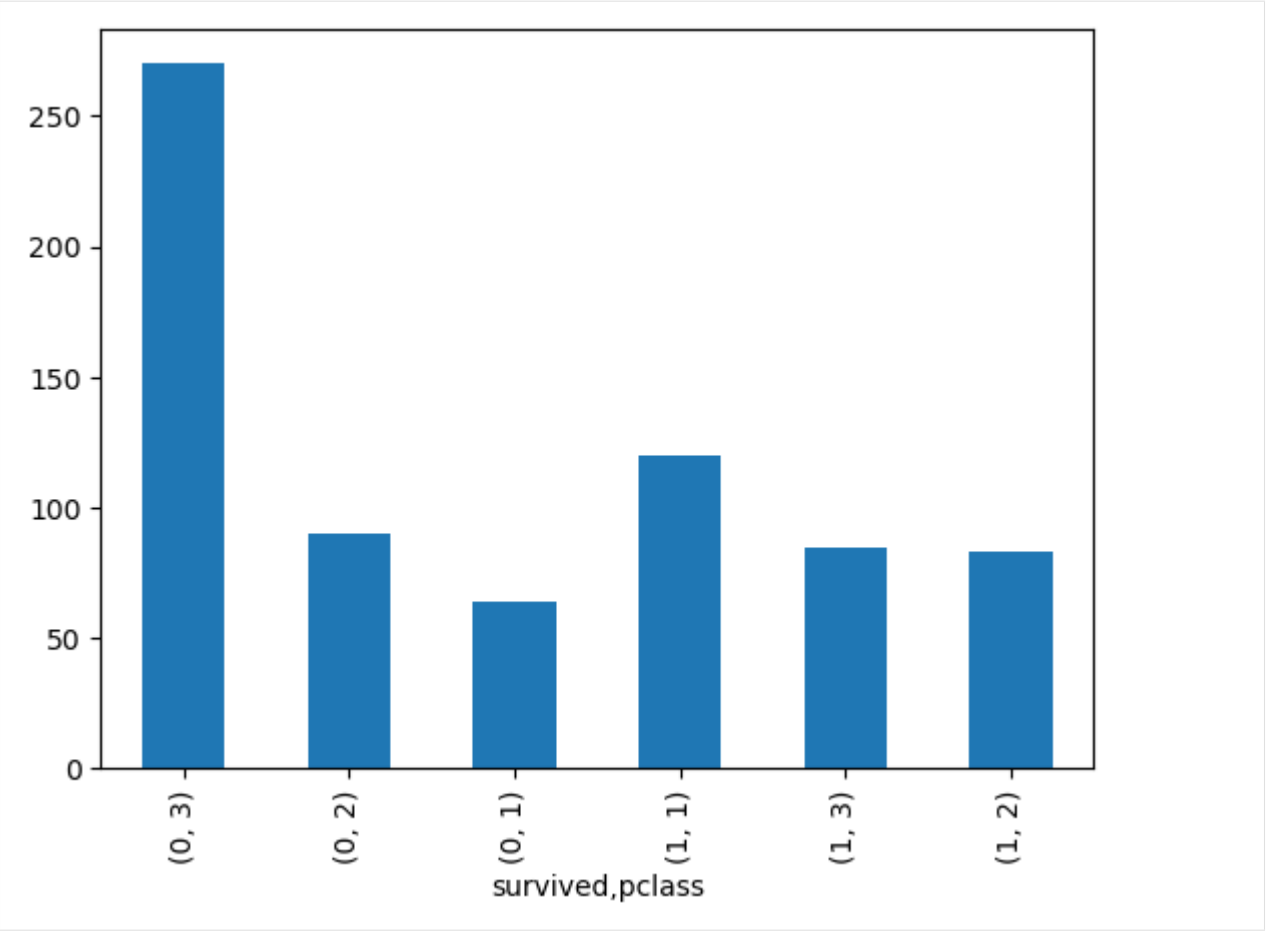
Or, we can inspect the distribution of each feature.

[43]:
```
titanic['age'].plot(kind='hist', bins=30)
plt.show()
```



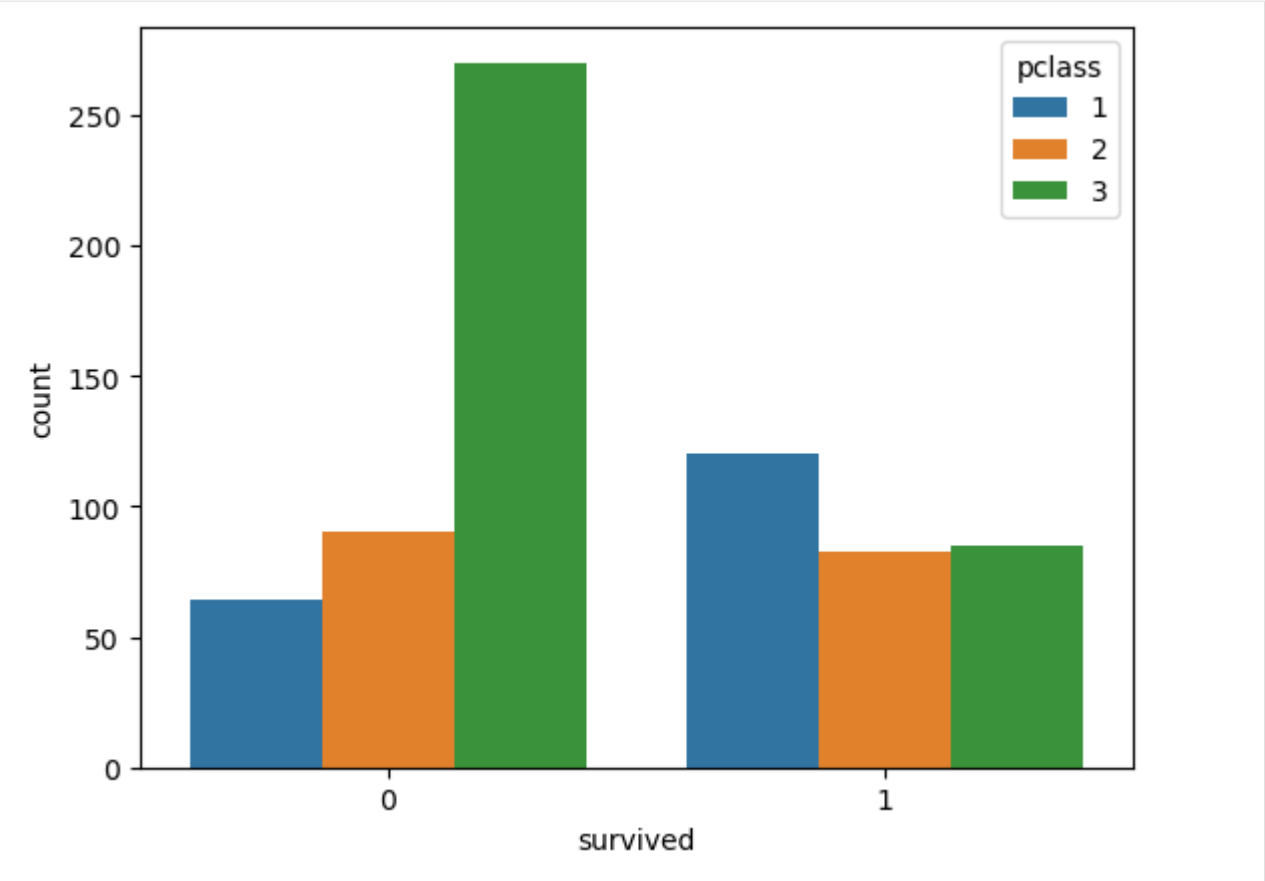
If we wish, we can use the `groupby` function in pandas, for instance, to show the counts for the `pclass` column grouped by `survived`.

[44]:
```
titanic.groupby('survived')['pclass'].value_counts().plot(kind="bar")
plt.show()
```

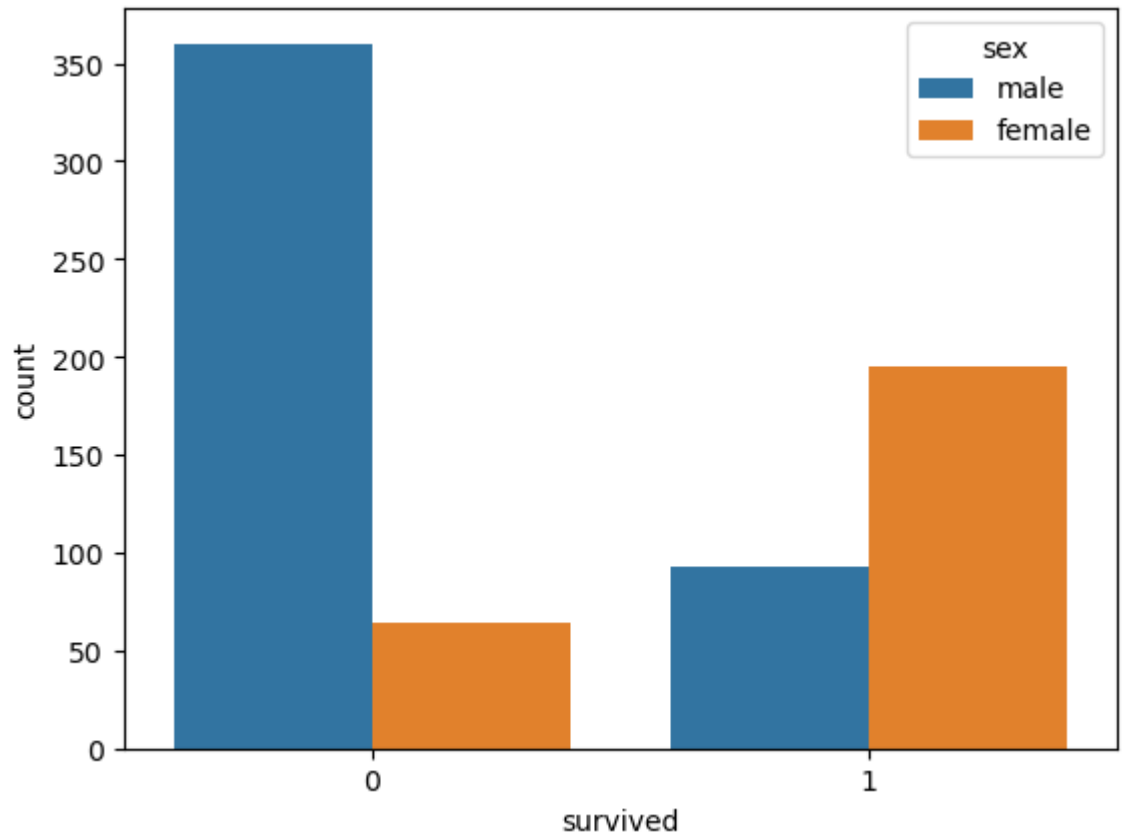


As we mentioned earlier, although the pandas library provides some functionality for plotting directly from DataFrames, there are other plotting libraries that provide improved graphs. Among the most popular is Seaborn. A few plots created with Seaborn are shown below.

[45]:
```
sns.countplot(data=titanic, x='survived', hue='pclass')
plt.show()
```

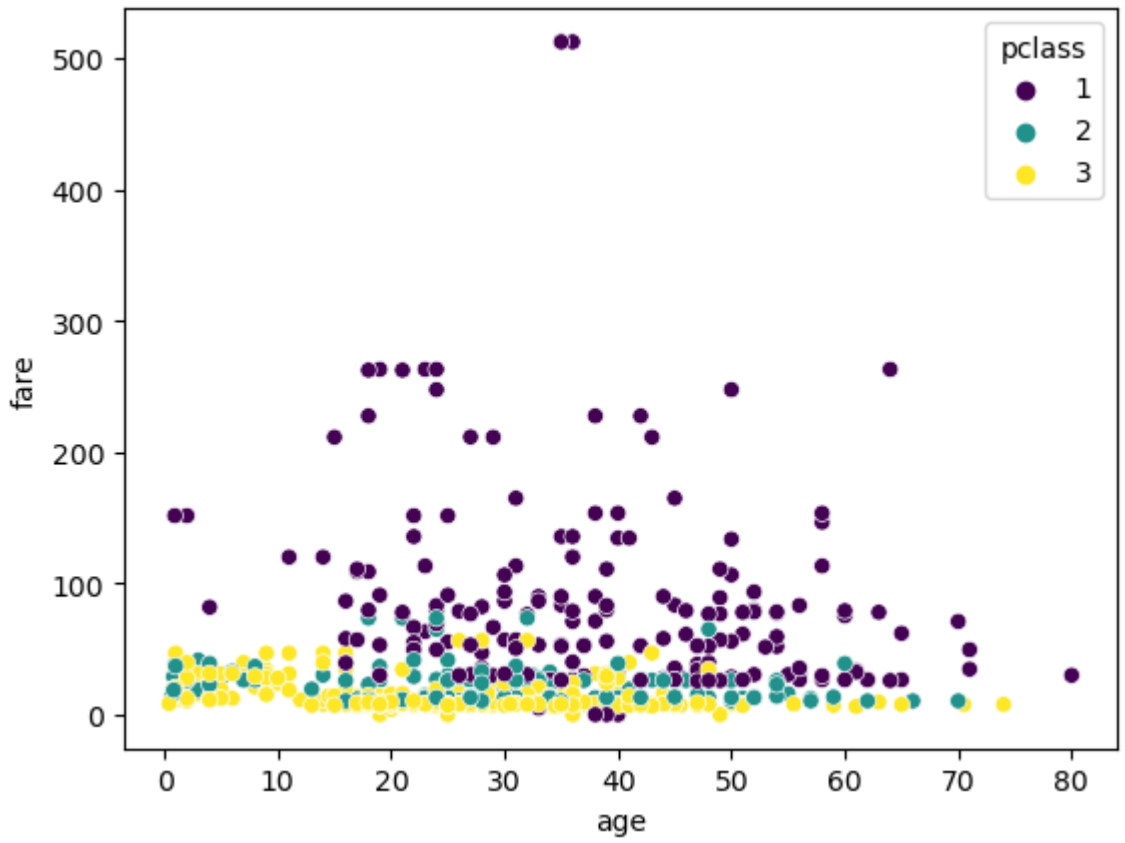


[46]:
```
sns.countplot(data=titanic, x='survived', hue='sex')
plt.show()
```



In the next figure we can see a scatter plot of the `age`, grouped by `fare` and `class`. As expected, the fare in the first class was more expensive, in comparison to the second and third classes.

[47]:
```
sns.scatterplot(data=titanic, x='age', y='fare', hue='pclass', palette='viridis')
plt.show()
```



Checking Correlated Features

Checking correlations between the features in a dataset can help to identify similarities between the features. If two features have high correlation, that means they contain similar or perhaps the same information, and if one of them is removed, the performance of the classification algorithm will be less affected.

We use the `corr()` method in pandas to check the correlation between the columns. Based on the correlation values, there aren't columns that are highly correlated. One of the reasons is that we already dropped several columns that were similar or the same as other columns.

[48]:
```
correlation = titanic.corr()
correlation
```

```
C:\Users\vakanski\AppData\Local\Temp\ipykernel_10716\825769680.py:1: FutureWarning: The␣
↪default value of numeric_only in DataFrame.corr is deprecated. In a future version, it␣
↪will default to False. Select only valid columns or specify the value of numeric_only␣
↪to silence this warning.
  correlation = titanic.corr()
```

[48]:

	index	survived	pclass	age	sibsp	parch	\
index	1.000000	0.029526	-0.035609	0.033681	-0.082704	-0.011672	
survived	0.029526	1.000000	-0.356462	-0.082446	-0.015523	0.095265	
pclass	-0.035609	-0.356462	1.000000	-0.365902	0.065187	0.023666	
age	0.033681	-0.082446	-0.365902	1.000000	-0.307351	-0.187896	
sibsp	-0.082704	-0.015523	0.065187	-0.307351	1.000000	0.383338	

(continues on next page)

(continued from previous page)

```
parch       -0.011672  0.095265  0.023666 -0.187896  0.383338  1.000000
fare         0.009655  0.266100 -0.552893  0.093143  0.139860  0.206624
adult_male   0.024069 -0.551151  0.094635  0.286543 -0.313016 -0.365580
alone        0.059677 -0.199741  0.150576  0.195766 -0.629408 -0.577109

                 fare  adult_male     alone
index        0.009655    0.024069  0.059677
survived     0.266100   -0.551151 -0.199741
pclass      -0.552893    0.094635  0.150576
age          0.093143    0.286543  0.195766
sibsp        0.139860   -0.313016 -0.629408
parch        0.206624   -0.365580 -0.577109
fare         1.000000   -0.177446 -0.262799
adult_male  -0.177446    1.000000  0.400718
alone       -0.262799    0.400718  1.000000
```

If we observe the correlation for the column `survived`, we can see that it is most correlated with `adult_male`, `pclass`, `fare`, and `alone`, and less correlated with `age`, `sibsp` (siblings), and `parch` (parent or child).

[49]:
```
correlation['survived']
```

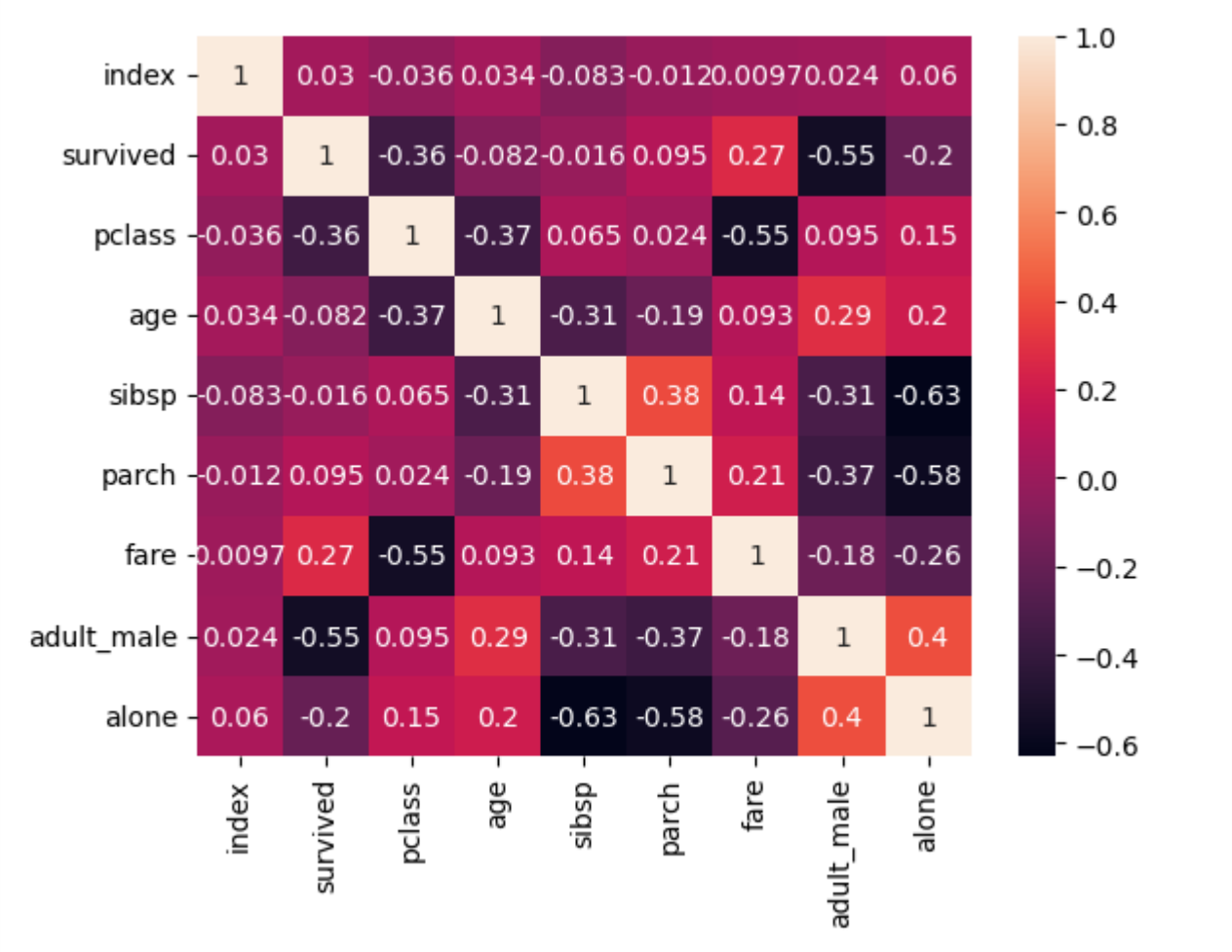
[49]:
```
index         0.029526
survived      1.000000
pclass       -0.356462
age          -0.082446
sibsp        -0.015523
parch         0.095265
fare          0.266100
adult_male   -0.551151
alone        -0.199741
Name: survived, dtype: float64
```

[50]:
```
correlation['fare']
```

[50]:
```
index         0.009655
survived      0.266100
pclass       -0.552893
age           0.093143
sibsp         0.139860
parch         0.206624
fare          1.000000
adult_male   -0.177446
alone        -0.262799
Name: fare, dtype: float64
```

The following heatmap shows the correlation between all features in a graph with a colorbar.

[51]:
```
sns.heatmap(correlation, annot=True)
plt.show()
```



7.11.2 11.2 Preprocessing Numerical Data

Tabular data can be classified into two main categories:

- **Numerical data**: a quantity represented by a real or integer number.
- **Categorical data**: a discrete value, typically represented by string labels taken from a finite list of possible choices, but it is also possible to be represented by numbers from a discrete set of possible choices.

Most machine learning algorithms are sensitive to the range of values that are used for numerical inputs, and expect the input features to be scaled before processing. **Feature scaling** is transforming the numerical features into a small range of values.

Common feature scaling techniques for numerical features include:

- Normalization
- Standardization
- Robust scaling

Whether or not a machine learning model requires scaling of the features depends on the model family. Linear models, such as logistic regression, generally benefit from scaling the features, while other models such as tree-based models (i.e., decision trees, random forests) do not need such preprocessing.

11.2.1 Normalization

Normalization is a scaling technique that transforms numerical features into a range of values between 0 and 1. When we work with features that have different ranges of values, normalizing the features can be a good practice. For example, if we have one feature (column) in the range from 100-1000, and another feature varies from 0.05-0.2, we can scale them so that they both have a range of values from 0 to 1.

Normalizing data is performed using the following formula, where X_{min} is the minimum value of feature X, and X_{max} is the maximum value of X.

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

For illustration purposes of normalization, we will use a smaller dataset called `tips` available in Seaborn.

[52]:
```
tip_data = sns.load_dataset('tips')
tip_data.head()
```

[52]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Let's separate all numerical features from the above data into a new DataFrame `original_features`.

[53]:
```
original_features = tip_data[['total_bill', 'tip', 'size']]
```

To perform normalization, we will use the `scikit-learn` library which provides the function `MinMaxScaler()` to scale the data to the range between 0 and 1. That is the default range, but we can also select an arbitrary range to scale the data.

The `fit` method in the code below first fits the data, i.e., for this task it calculates the minimum and maximum values for each column.

Afterwards, the `transform` method scales the data, i.e., it uses the calculated minimum and maximum values for each column and substitutes them in the above formula to obtain scaled values of the data.

The syntax is `scaler.fit(data)` and `scaler.transform(data)`.

[54]:
```
from sklearn.preprocessing import MinMaxScaler

minmax_scaler = MinMaxScaler()

minmax_scaler.fit(original_features)

normalized_features = minmax_scaler.transform(original_features)
```

[55]:
```
# Show the first five rows in 'total_bill', 'tip', and 'size'
normalized_features[:5]
```

[55]:
```
array([[0.29157939, 0.00111111, 0.2       ],
       [0.1522832 , 0.07333333, 0.4       ],
       [0.3757855 , 0.27777778, 0.4       ],
       [0.43171345, 0.25666667, 0.2       ],
       [0.45077503, 0.29      , 0.6       ]])
```

The `scikit-learn` library also provides a combined method `fit_transform` which calls first `fit` and then `transform` in one step. This can be more efficient than calling `fit` and `transform` separately. The syntax is `scaler.fit_transform(data)`.

[56]:
```
normalized_features_2 = minmax_scaler.fit_transform(original_features)
```

[57]:
```
# Show the first five rows in 'total_bill', 'tip', and 'size'
normalized_features_2[:5]
```

[57]:
```
array([[0.29157939, 0.00111111, 0.2       ],
       [0.1522832 , 0.07333333, 0.4       ],
       [0.3757855 , 0.27777778, 0.4       ],
       [0.43171345, 0.25666667, 0.2       ],
       [0.45077503, 0.29      , 0.6       ]])
```

The output of `MinMaxScaler()` is a NumPy array, with the values in each column scaled in the range between 0 and 1.

11.2.2 Standardization

Standardization is another scaling technique where numerical features are rescaled to have 0 mean (μ) and 1 standard deviation (σ).

The formula for standardization is as follows:

$$X_{std} = \frac{X - \mu}{\sigma}$$

where X_{std} is the standardized feature, X is the original feature, μ is the mean of the feature, and σ is the standard deviation.

When should we standardize the features? When we know that the training data has a normal (Gaussian) distribution. And, if the data does not have normal distribution, then normalization is a preferable scaling technique to standardization.

With some machine learning algorithms the performance will be the same with features scaling using normalization or standardization, but with other algorithms, there can be a difference in the performance. Therefore, in some cases, we can try both feature scaling techniques, especially if we are not sure about the distribution of the data.

Standardization is implemented in `scikit-learn` with `StandardScaler`. Similar to `MinMaxScaler`, we can either use the syntax `scaler.fit_transform(data)` or the syntax with `scaler.fit(data)` and `scaler.transform(data)`. And, we will explain the difference between these two syntaxes in the next lectures when we will introduce the concepts of training and testing datasets.

[58]:
```
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler()

standardized_features = std_scaler.fit_transform(original_features)
```

[59]:
```
standardized_features[:5]
```

[59]:
```
array([[-0.31471131, -1.43994695, -0.60019263],
       [-1.06323531, -0.96920534,  0.45338292],
       [ 0.1377799 ,  0.36335554,  0.45338292],
       [ 0.4383151 ,  0.22575414, -0.60019263],
       [ 0.5407447 ,  0.4430195 ,  1.50695847]])
```

We can inspect the mean and standard variation of the original data using the `mean_` and `var_` attributes. The convention in scikit-learn is that if an attribute is learned from the data, its name ends with an underscore, as in `mean_` and `var_` for the `StandardScaler`.

[60]:
```
# The mean of each feature in the original data
std_scaler.mean_
```

[60]:
```
array([19.78594262,  2.99827869,  2.56967213])
```

[61]:
```
# The variance each feature in the original data
std_scaler.var_
```

[61]:
```
array([78.92813149,  1.90660851,  0.9008835 ])
```

As expected, each column in the scaled data `standardized_features` has zero mean and unit variance.

[62]:
```
# The mean of each feature in the scaled data
np.round(standardized_features.mean(axis=0))
```

[62]:
```
array([-0.,  0., -0.])
```

[63]:
```
# The variance each feature in the scaled data
np.round(standardized_features.std(axis=0))
```

[63]:
```
array([1., 1., 1.])
```

It is easy to confuse normalization with standardization, since standardization rescales the data to have a normal distribution with mean 0 and standard deviation 1, therefore, pay attention and try not to confuse these two scaling techniques.

11.2.3 Robust Scaling

Scikit-learn provides another scaling method called `RobustScaler`, which is more suitable when the data contain many outliers.

`RobustScaler` applies similar scaling to standardization, but it uses the median and the interquartile range (IQR) instead of the mean and standard deviation to scale the features. This makes it less sensitive to extreme values or outliers in the data. Recall the Interquartile Range(IQR) is the difference between the 1st quartile (25th percentile) and the 3rd quartile (75th percentile).

Because of that, `RobustScaler` is also more suitable for datasets with non-normal distributions.

[64]:
```
from sklearn.preprocessing import RobustScaler

rob_scaler = RobustScaler()
robust_scaled_features = rob_scaler.fit_transform(original_features)
```

[65]:
```
robust_scaled_features[:5]
```

[65]:
```
array([[-0.07467532, -1.2096    ,  0.        ],
       [-0.69155844, -0.7936    ,  1.        ],
       [ 0.29823748,  0.384     ,  1.        ],
       [ 0.54591837,  0.2624    ,  0.        ],
       [ 0.63033395,  0.4544    ,  2.        ]])
```

We can confirm that the columns in the scaled data have 0 median.

[66]:
```
np.round(np.median(robust_scaled_features, axis=0))
```

[66]:
```
array([-0.,  0.,  0.])
```

The `MinMaxScaler`, `StandardScaler`, and `RobustScaler` in scikit-learn are also called **transformers**, since they are used to perform various data transformations on the original dataset before feeding the data into a machine learning model. Transformers are an essential part of the data processing in scikit-learn's pipeline.

7.11.3 11.3 Preprocessing Categorical Data

Categorical data contain a limited number of discrete categories. An example is the feature `who` in the `titanic` dataset that has three categories: `man`, `woman`, and `child`.

In many cases, categorical features have text values, and they need to be converted into numerical values in order to be processed by machine learning algorithms.

We will look into the following techniques for converting categorical features into numerical features:

- Mapping method
- Ordinal encoding
- Label encoding
- Pandas dummies
- One-hot encoding

The first three techniques produce a single number for each category, and the last two techniques produce one-hot matrix.

We are going to use again the Titanic dataset, which has several categorical features.

[67]:
```
titanic.head()
```

[67]:

	index	survived	pclass	sex	age	sibsp	parch	fare	who	adult_male	embark_town	alone
0	0	0	3	male	22.0	1	0	7.2500	man	True	Southampton	False
1	1	1	1	female	38.0	1	0	71.2833	woman	False	Cherbourg	False
2	2	1	3	female	26.0	0	0	7.9250	woman	False	Southampton	True
3	3	1	1	female	35.0	1	0	53.1000	woman	False	Southampton	False
4	4	0	3	male	35.0	0	0	8.0500	man	True	Southampton	True

11.3.1 Mapping Method

The mapping method is a straightforward way to encode categorical features when there are few categories. For instance, for the `who` feature, we will create a dictionary `map_dict` whose keys are the three categories `man`, `woman`, `child`, and they are mapped to numerical values 0, 1, and 2.

[68]:
```
titanic['who'].value_counts()
```

[68]:
```
man      413
woman    216
child     83
Name: who, dtype: int64
```

[69]:
```
map_dict = {'man': 0, 'woman': 1, 'child': 2}
```

The `map()` method is applied next to map the keys to values in the `who` column.

[70]:
```
titanic['who'] = titanic['who'].map(map_dict)
```

Now the `who` feature is numerical, and all instances with the class `man` were replaced with `0`, and the same applies to the other two classes.

[71]:
```
titanic.head()
```

[71]:

	index	survived	pclass	sex	age	sibsp	parch	fare	who	adult_male	embark_town	alone
0	0	0	3	male	22.0	1	0	7.2500	0	True	Southampton	False
1	1	1	1	female	38.0	1	0	71.2833	1	False	Cherbourg	False
2	2	1	3	female	26.0	0	0	7.9250	1	False	Southampton	True
3	3	1	1	female	35.0	1	0	53.1000	1	False	Southampton	False
4	4	0	3	male	35.0	0	0	8.0500	0	True	Southampton	True

[72]:
```
# verify that value_counts remained the same after the mapping
titanic['who'].value_counts()
```

[72]:
```
0    413
1    216
2     83
Name: who, dtype: int64
```

11.3.2 Ordinal Encoding

Ordinal encoding can be implemented with the `OrdinalEncoder` in scikit-learn, which will automatically encode each category with a different numerical value. This method is often preferred, since it is automated and less prone to errors.

Let's apply it to the columns `alone` and `adult_male`.

[73]:
```
titanic['alone'].value_counts()
```

[73]:
```
True     402
False    310
Name: alone, dtype: int64
```

[74]:
```
titanic['adult_male'].value_counts()
```

[74]:
```
True     413
False    299
Name: adult_male, dtype: int64
```

[75]:
```
from sklearn.preprocessing import OrdinalEncoder

categs_feats = titanic[['adult_male', 'alone']]

encoder = OrdinalEncoder()

categs_encoded = encoder.fit_transform(categs_feats)
```

The output of the `OrdinalEncoder` is a NumPy array `categs_encoded`, shown below.

[76]:
```
categs_encoded
```

[76]:
```
array([[1., 0.],
       [0., 0.],
       [0., 1.],
       ...,
       [0., 1.],
       [1., 1.],
       [1., 1.]])
```

In the next cell, we will convert the NumPy array `categs_encoded` into pandas DataFrame. In this line, `columns=categs_feats.columns` specifies the column names for the DataFrame, and `index=categs_feats.index` specifies the row index for the DataFrame.

Note below that the text values in the columns `alone` and `adult_male` have been replaced with numeric values `0` or `1`.

[77]:
```
titanic[['adult_male', 'alone']] = pd.DataFrame(categs_encoded, columns=categs_feats.
↪columns, index=categs_feats.index)
titanic.head()
```

[77]:

	index	survived	pclass	sex	age	sibsp	parch	fare	who	\
0	0	0	3	male	22.0	1	0	7.2500	0	
1	1	1	1	female	38.0	1	0	71.2833	1	
2	2	1	3	female	26.0	0	0	7.9250	1	
3	3	1	1	female	35.0	1	0	53.1000	1	
4	4	0	3	male	35.0	0	0	8.0500	0	

(continues on next page)

(continued from previous page)

```
   adult_male  embark_town  alone
0         1.0  Southampton    0.0
1         0.0    Cherbourg    0.0
2         0.0  Southampton    1.0
3         0.0  Southampton    0.0
4         1.0  Southampton    1.0
```

[78]:
```
# verify that value_counts remained the same after the mapping
titanic['alone'].value_counts()
```

[78]:
```
1.0    402
0.0    310
Name: alone, dtype: int64
```

We can also check the applied mapping between the categories and the numerical values via the attribute `categories_`.

[79]:
```
encoder.categories_
```

[79]:
```
[array([False,  True]), array([False,  True])]
```

Note that `OrdinalEncoder` can not handle missing values, and if we try to apply it to a column with missing values, we will get an error.

Also, we need to be careful when applying this encoding strategy, because by default, `OrdinalEncoder` uses a lexicographical strategy to map string category labels to integers. For instance, suppose the dataset has a categorical variable named `"size"` with categories such as "S", "M", "L", "XL", and we would like the integer representation to respect the meaning of the sizes by mapping them to increasing integers such as `0, 1, 2, 3`. However, the lexicographical strategy used by default would map the labels "S", "M", "L", "XL" to 2, 1, 0, 3, by following the alphabetical order. To avoid that, we can pass a list with the expected order for the `categories` argument for each feature (e.g., `encoder = OrdinalEncoder(categories=[True, False]` for the column `alone`).

If a categorical variable does not carry any meaningful order information, then we can consider using one-hot encoding described in the sections below.

11.3.3 Label Encoding

Label encoding is used to encode categorical values in the target label column with the `LabelEncoder` in scikit-learn.

In this case, the target label column `survived` has numerical values, and it does not need to be encoded. Therefore, let's apply the `LabelEncoder` to the `embark_town` column.

[80]:
```
from sklearn.preprocessing import LabelEncoder

embtown_feat = titanic[['embark_town']]

label_encoder = LabelEncoder()

embtown_encoded = label_encoder.fit_transform(embtown_feat)
```

```
C:\Users\vakanski\anaconda3\Lib\site-packages\sklearn\preprocessing\_label.py:114:␣
↪DataConversionWarning: A column-vector y was passed when a 1d array was expected.␣
↪Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

The output of Label Encoder is also a NumPy array. Let's convert it to a pandas dataframe, and add it as a new column embark_town_ord.

[81]:
```
titanic['embark_town_ord'] = pd.DataFrame(embtown_encoded, columns=embtown_feat.columns,␣
↪index=embtown_feat.index)

titanic.head()
```

[81]:
```
   index  survived  pclass     sex   age  sibsp  parch     fare  who  \
0      0         0       3    male  22.0      1      0   7.2500    0
1      1         1       1  female  38.0      1      0  71.2833    1
2      2         1       3  female  26.0      0      0   7.9250    1
3      3         1       1  female  35.0      1      0  53.1000    1
4      4         0       3    male  35.0      0      0   8.0500    0

   adult_male  embark_town  alone  embark_town_ord
0         1.0  Southampton    0.0                2
1         0.0    Cherbourg    0.0                0
2         0.0  Southampton    1.0                2
3         0.0  Southampton    0.0                2
4         1.0  Southampton    1.0                2
```

[82]:
```
label_encoder.classes_
```

[82]:
```
array(['Cherbourg', 'Queenstown', 'Southampton'], dtype=object)
```

[83]:
```
titanic['embark_town_ord'].value_counts()
```

[83]:
```
2    554
0    130
1     28
Name: embark_town_ord, dtype: int64
```

11.3.4 Pandas Dummies

Pandas provides a function `get_dummies` that can be also used to handle categorical features. This function creates new columns based on the number of available categories in a target column. For example, let's apply it to the feature `sex`.

[84]:
```
titanic.head()
```

[84]:
```
   index  survived  pclass     sex   age  sibsp  parch     fare  who  \
0      0         0       3    male  22.0      1      0   7.2500    0
1      1         1       1  female  38.0      1      0  71.2833    1
2      2         1       3  female  26.0      0      0   7.9250    1
3      3         1       1  female  35.0      1      0  53.1000    1
4      4         0       3    male  35.0      0      0   8.0500    0

   adult_male  embark_town  alone  embark_town_ord
0         1.0  Southampton    0.0                2
1         0.0    Cherbourg    0.0                0
2         0.0  Southampton    1.0                2
3         0.0  Southampton    0.0                2
4         1.0  Southampton    1.0                2
```

[85]:
```
dummies = pd.get_dummies(titanic['sex'])
```

[86]:
```
titanic = pd.concat([titanic.drop('sex',axis=1),dummies], axis=1)
```

Note that new columns `female` and `male` with 0 or 1 values were added to the right of the DataFrame.

[87]:
```
titanic.head()
```

[87]:
```
   index  survived  pclass   age  sibsp  parch     fare  who  adult_male  \
0      0         0       3  22.0      1      0   7.2500    0         1.0
1      1         1       1  38.0      1      0  71.2833    1         0.0
2      2         1       3  26.0      0      0   7.9250    1         0.0
3      3         1       1  35.0      1      0  53.1000    1         0.0
4      4         0       3  35.0      0      0   8.0500    0         1.0

   embark_town  alone  embark_town_ord  female  male
0  Southampton    0.0                2       0     1
1    Cherbourg    0.0                0       1     0
2  Southampton    1.0                2       1     0
3  Southampton    0.0                2       1     0
4  Southampton    1.0                2       0     1
```

This type of encoding is also called one-hot encoding, where each category (unique value) in the column `sex` became a column, and for each row (sample), 1 specifies the category to which it belongs.

11.3.5 One-Hot Encoding

Scikit-learn provides a function `OneHotEncoder` that converts a feature into one-hot matrix. As with the pandas dummies, additional columns corresponding to the values of the given categories are created.

Let's apply it to `embark_town`.

[88]:
```
titanic['embark_town'].value_counts()
```

[88]:
```
Southampton    554
Cherbourg      130
Queenstown      28
Name: embark_town, dtype: int64
```

[89]:
```
titanic.head()
```

[89]:
```
   index  survived  pclass   age  sibsp  parch     fare  who  adult_male  \
0      0         0       3  22.0      1      0   7.2500    0         1.0
1      1         1       1  38.0      1      0  71.2833    1         0.0
2      2         1       3  26.0      0      0   7.9250    1         0.0
3      3         1       1  35.0      1      0  53.1000    1         0.0
4      4         0       3  35.0      0      0   8.0500    0         1.0

   embark_town  alone  embark_town_ord  female  male
0  Southampton    0.0                2       0     1
1    Cherbourg    0.0                0       1     0
2  Southampton    1.0                2       1     0
3  Southampton    0.0                2       1     0
4  Southampton    1.0                2       0     1
```

[90]:
```
from sklearn.preprocessing import OneHotEncoder

one_hot = OneHotEncoder()

town_encoded = one_hot.fit_transform(titanic[['embark_town']])
```

[91]:
```
one_hot.categories_
```

[91]:
```
[array(['Cherbourg', 'Queenstown', 'Southampton'], dtype=object)]
```

[92]:
```
town_encoded
```

[92]:
```
<712x3 sparse matrix of type '<class 'numpy.float64'>'
        with 712 stored elements in Compressed Sparse Row format>
```

The output of OneHotEncoder is a sparse matrix. We will need to convert it into NumPy array first, and afterward we can convert it into pandas DataFrame.

[93]:
```
town_encoded = town_encoded.toarray()
```

[94]:
```
columns = list(one_hot.categories_)

town_df = pd.DataFrame(town_encoded, columns=columns)

town_df.head()
```

[94]:

	Cherbourg	Queenstown	Southampton
0	0.0	0.0	1.0
1	1.0	0.0	0.0
2	0.0	0.0	1.0
3	0.0	0.0	1.0
4	0.0	0.0	1.0

[95]:
```
titanic.drop('embark_town',axis=1, inplace=True)
```

[96]:
```
titanic[['Cherbourg', 'Queenstown', 'Southampton']] = town_df
```

[97]:
```
titanic.head()
```

[97]:

	index	survived	pclass	age	sibsp	parch	fare	who	adult_male	alone	embark_town_ord	female	male	Cherbourg	Queenstown	Southampton
0	0	0	3	22.0	1	0	7.2500	0	1.0	0.0	2	0	1	0.0	0.0	1.0
1	1	1	1	38.0	1	0	71.2833	1	0.0	0.0	0	1	0	1.0	0.0	0.0
2	2	1	3	26.0	0	0	7.9250	1	0.0	1.0	2	1	0	0.0	0.0	1.0
3	3	1	1	35.0	1	0	53.1000	1	0.0	0.0	2	1	0	0.0	0.0	1.0
4	4	0	3	35.0	0	0	8.0500	0	1.0	1.0	2	0	1	0.0	0.0	1.0

Choosing an encoding strategy

Choosing an encoding strategy depends on the used models and the type of categories (i.e., ordinal vs. nominal). In general, One-Hot Encoding is the encoding strategy used when the downstream models are linear models, while Ordinal Encoding is often a good strategy with tree-based models.

With ordinal encoding, there is an order in the resulting categories, e.g. `0 < 1 < 2` (called **ordinal categories**). The impact of violating this ordering assumption is dependent on the downstream models. Linear models will be impacted by misordered categories, while tree-based models will not.

One-hot encoding is applied when the ordering of the categories is not important. Such categories are also called **nominal categories**. This encoding can cause computational inefficiency in tree-based models with high number of categories, and because of this, it is not recommended to use with these models.

7.11.4 11.4 Combining Numerical and Categorical Features

Now let's prepare the numerical and categorical data in the `titanic` dataset and train a classification model.

First, assign the `survived` column to be the target label `y`.

[98]:
```
y = titanic['survived']
```

We will use the other columns to be data features `X`, therefore let's drop the `survived` column and the `index` column.

It is very important to always remove the `index` column from the data used for training a model. If we leave the `index` column in the data, the model can learn to associate the target labels with the index of each data point (row).

[99]:
```
X = titanic.drop(['survived', 'index'], axis=1)
```

[100]:
```
X
```

[100]:
```
     pclass   age  sibsp  parch     fare  who  adult_male  alone  \
0         3  22.0      1      0   7.2500    0         1.0    0.0
1         1  38.0      1      0  71.2833    1         0.0    0.0
2         3  26.0      0      0   7.9250    1         0.0    1.0
3         1  35.0      1      0  53.1000    1         0.0    0.0
4         3  35.0      0      0   8.0500    0         1.0    1.0
..      ...   ...    ...    ...      ...  ...         ...    ...
707       3  39.0      0      5  29.1250    1         0.0    0.0
708       2  27.0      0      0  13.0000    0         1.0    1.0
709       1  19.0      0      0  30.0000    1         0.0    1.0
710       1  26.0      0      0  30.0000    0         1.0    1.0
711       3  32.0      0      0   7.7500    0         1.0    1.0

     embark_town_ord  female  male  Cherbourg  Queenstown  Southampton
0                  2       0     1        0.0         0.0          1.0
1                  0       1     0        1.0         0.0          0.0
2                  2       1     0        0.0         0.0          1.0
3                  2       1     0        0.0         0.0          1.0
4                  2       0     1        0.0         0.0          1.0
..               ...     ...   ...        ...         ...          ...
707                1       1     0        0.0         1.0          0.0
708                2       0     1        0.0         0.0          1.0
709                2       1     0        0.0         0.0          1.0
710                0       0     1        1.0         0.0          0.0
```

(continues on next page)

(continued from previous page)

```
711                1       0     1        0.0         1.0          0.0

[712 rows x 14 columns]
```

We will first split the dataset into a training and test dataset. This step will be explained in more detail in the next lectures. The objective of this lecture is to learn how to preprocess the data and prepare it for model fitting.

[101]:
```
from sklearn.model_selection import train_test_split

# split into train & test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=123, stratify=y)
```

[102]:
```
print('Training data inputs', X_train.shape)
print('Training labels', y_train.shape)
print('Testing data inputs', X_test.shape)
print('Testing labels', y_test.shape)
```

```
Training data inputs (534, 14)
Training labels (534,)
Testing data inputs (178, 14)
Testing labels (178,)
```

We will apply standard scaling to the training and test datasets.

[103]:
```
# Apply StandardScaler
scaler = StandardScaler()

scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Now, let's apply k-Nearest Neighbor classifier from the scikit-learn library. The function `fit()` is used to fit the model to the data.

[104]:
```
from sklearn import neighbors

# create the model
knn_model = neighbors.KNeighborsClassifier(n_neighbors=5)

# fit the model
knn_model.fit(X_train_scaled, y_train)
```

[104]:
```
KNeighborsClassifier()
```

The function `score()` is used to predict the accuracy of the model. The model can predict whether a passenger survived with an accuracy of 82.58%.

[105]:
```
# score on test set
accuracy = knn_model.score(X_test_scaled, y_test)
print('The test accuracy of k-Nearest Neighbors is {0:5.2f} %'.format(accuracy*100))
```

```
The test accuracy of k-Nearest Neighbors is 82.58 %
```

Next, let's use the trained model to predict the `survived` labels for the first 10 passengers in the `X_test` dataset.

[106]:
```
# Make predictions on the test data
y_pred = knn_model.predict(X_test_scaled[:10])
```

[107]:
```
# Show the predictions
y_pred
```

[107]:
```
array([0, 0, 0, 1, 1, 1, 1, 0, 1, 0], dtype=int64)
```

[108]:
```
# Show the actual values from the 'survived' column
np.array(y_test[:10])
```

[108]:
```
array([0, 1, 0, 0, 0, 1, 1, 0, 1, 0], dtype=int64)
```

As we can see, for the first 10 samples, the model predicted correctly the target label for 7 samples.

We will have a separate lecture on `scikit-learn` in which we will explain in more detail how to perform classification with machine learning models.

7.11.5 References

1. Complete Machine Learning Package, Jean de Dieu Nyandwi, available at: https://github.com/Nyandwi/machine_learning_complete.
2. Advanced Python for Data Science, University of Cincinnati, available at: https://github.com/uc-python/advanced-python-datasci.
3. Python Machine Learning (2nd Ed.) Code Repository, Sebastian Raschka, available at: https://github.com/rasbt/python-machine-learning-book-2nd-edition.

BACK TO TOP

7.12 Lecture 12 - Data Visualization with Seaborn

Seaborn is a Python library for data visualization that provides an interface for drawing plots and conducting data exploration via visualization and informative graphics. Internally, Seaborn uses Matplotlib to create the plots, therefore it can be considered a high-level interface for Matplotlib which allows to quickly and easily customize our plots.

For more information please visit the official website. Examples of plots created with Seaborn can be found in the gallery page.

By convention, Seaborn is imported as `sns`. The name Seaborn was originally based on a character named Samuel Norman Seaborn from a television show, and the alias `sns` is based on the character's initials.

To explain the functionality of Seaborn, in this lecture we will use the following four datasets: `titanic`, `fmri`, `tips`, and `flights`, which can be loaded directly as DataFrames from the Seaborn datasets. We already worked with the `titanic` and `tips` datasets in previous lectures.

The first few rows of these datasets are shown below.

[1]:
```
# Import libraries
import seaborn as sns
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Loading the datasets for this lecture
titanic = sns.load_dataset('titanic')
fmri = sns.load_dataset('fmri')
tips = sns.load_dataset('tips')
flights = sns.load_dataset('flights')
```

[2]:
```
titanic.head(3)
```

[2]:
```
   survived  pclass     sex   age  sibsp  parch     fare embarked  class  \
0         0       3    male  22.0      1      0   7.2500        S  Third
1         1       1  female  38.0      1      0  71.2833        C  First
2         1       3  female  26.0      0      0   7.9250        S  Third

     who  adult_male deck  embark_town alive  alone
0    man        True  NaN  Southampton    no  False
1  woman       False    C    Cherbourg   yes  False
2  woman       False  NaN  Southampton   yes   True
```

[3]:
```
fmri.head(3)
```

[3]:
```
  subject  timepoint event    region    signal
0     s13         18  stim  parietal -0.017552
1      s5         14  stim  parietal -0.080883
2     s12         18  stim  parietal -0.081033
```

[4]:
```
tips.head(3)
```

[4]:
```
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
```

[5]:
```
flights.head(3)
```

[5]:
```
   year month  passengers
0  1949   Jan         112
1  1949   Feb         118
2  1949   Mar         132
```

7.12.1 12.1 Relational Plots

Relational plots are used to visualize the statistical relationship between the features in a dataset.

We will provide examples using the following relational plots in Seaborn:

- Line Plots
- Scatter Plots

Line Plots

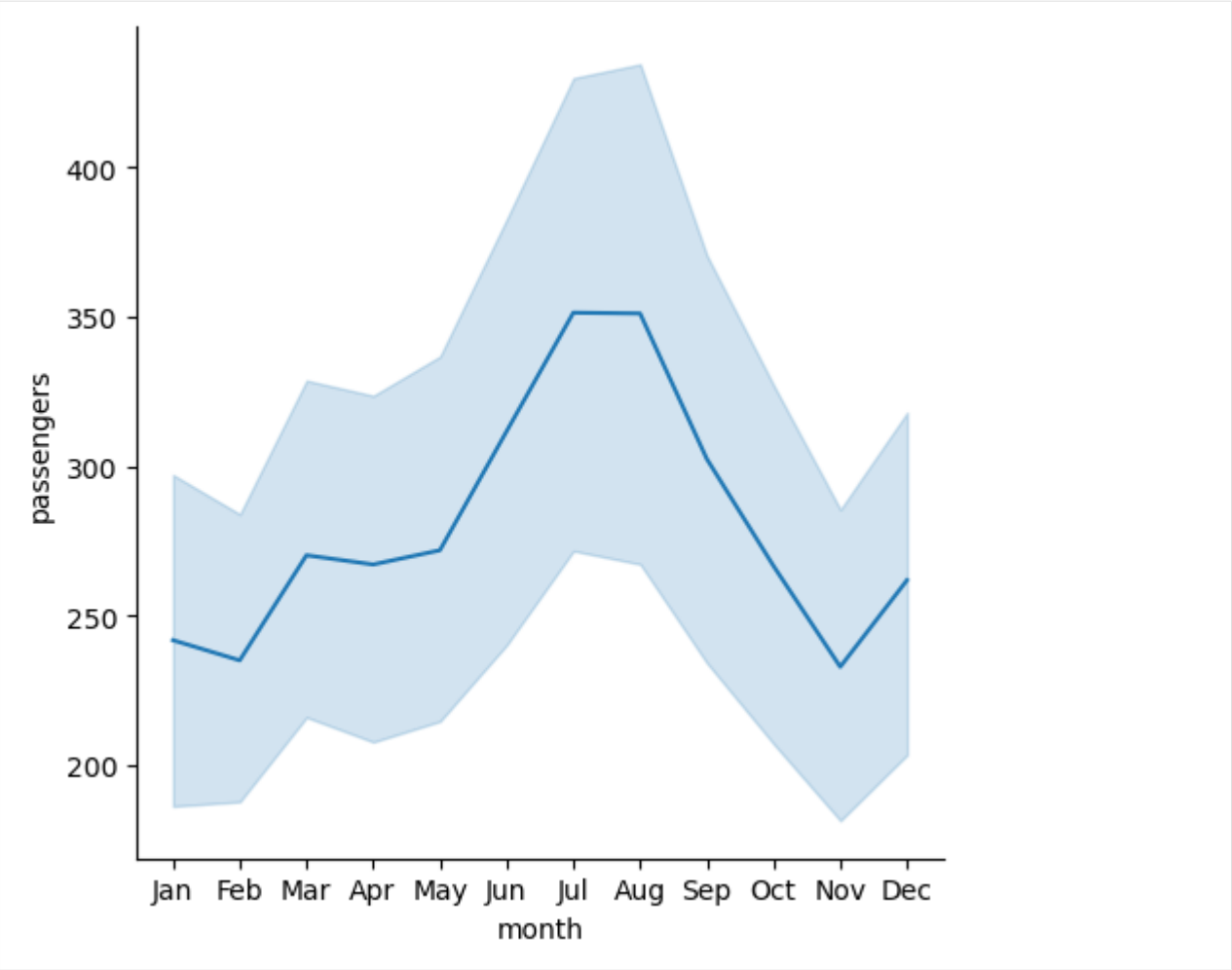
Line plots can be created in Seaborn by using the function `relplot()`, with the `kind` parameter set to `line`. Similarly, setting the `kind` to `scatter` is used for creating scatter plots. In general, `relplot()` is a flexible function that allows to visualize different relational plots between the features.

In the figure below, we used the `flights` dataset, and plotted the feature `month` along the x-axis and the `passenger` along the y-axis. The plot also calculates the confidence intervals and draws error bars representing the uncertainty for the number of passengers per month. Many plotting functions in Seaborn use statistical estimators to plot data statistics by default.

Note also that when we create Seaborn plots in Jupyter notebooks, the figure object information is displayed above the plot, in this case `<seaborn.axisgrid.FacetGrid at 0x20dd5e50810>`. The same is true for Matplotlib plots, as they also return the figure object information in Jupyter notebooks. We can suppress the text by either adding `plt.show()` in the cells, or by adding a semicolon to the last line in the cells. We will add a semicolon for all plots in this lecture to suppress the text for the figure objects.

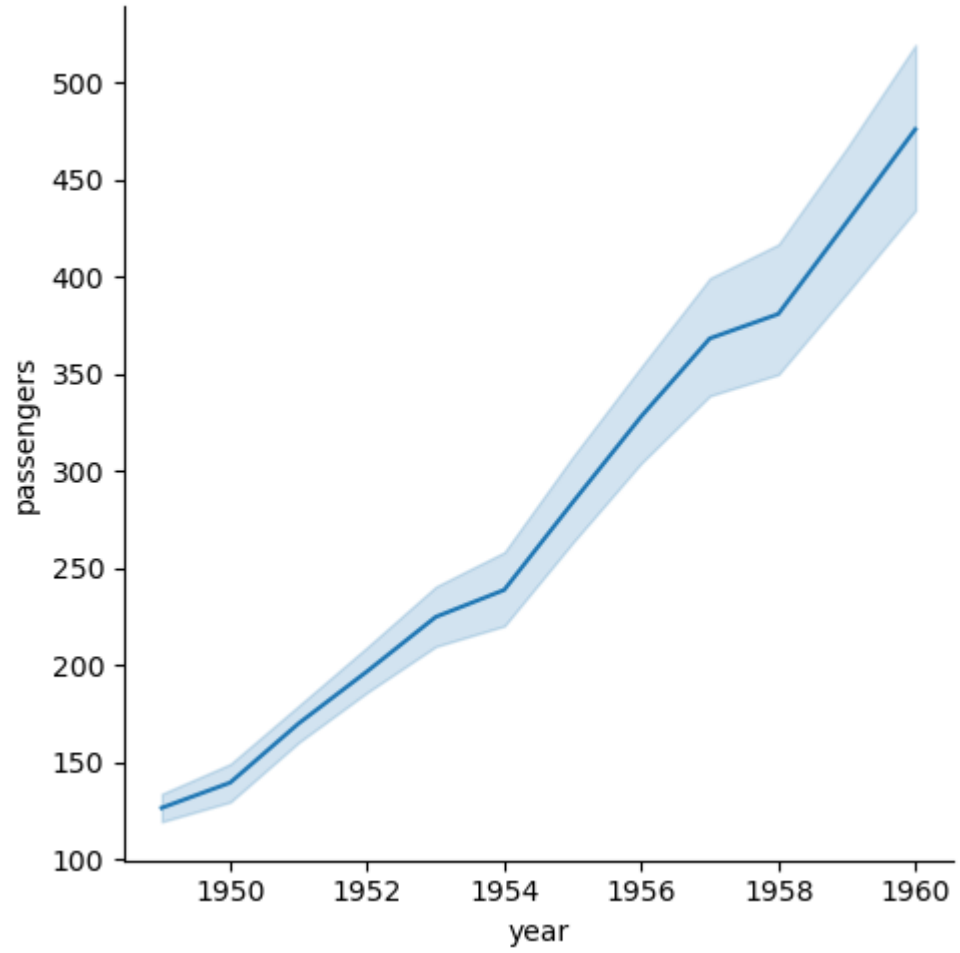
[6]:
```
sns.relplot(data=flights, x='month', y='passengers', kind='line')
# Use either 'plt.show()' or add a semicolon to suppress the returned text above the plot
```

[6]:
```
<seaborn.axisgrid.FacetGrid at 0x20dd5e50810>
```



And another plot shows the number of passengers per year.

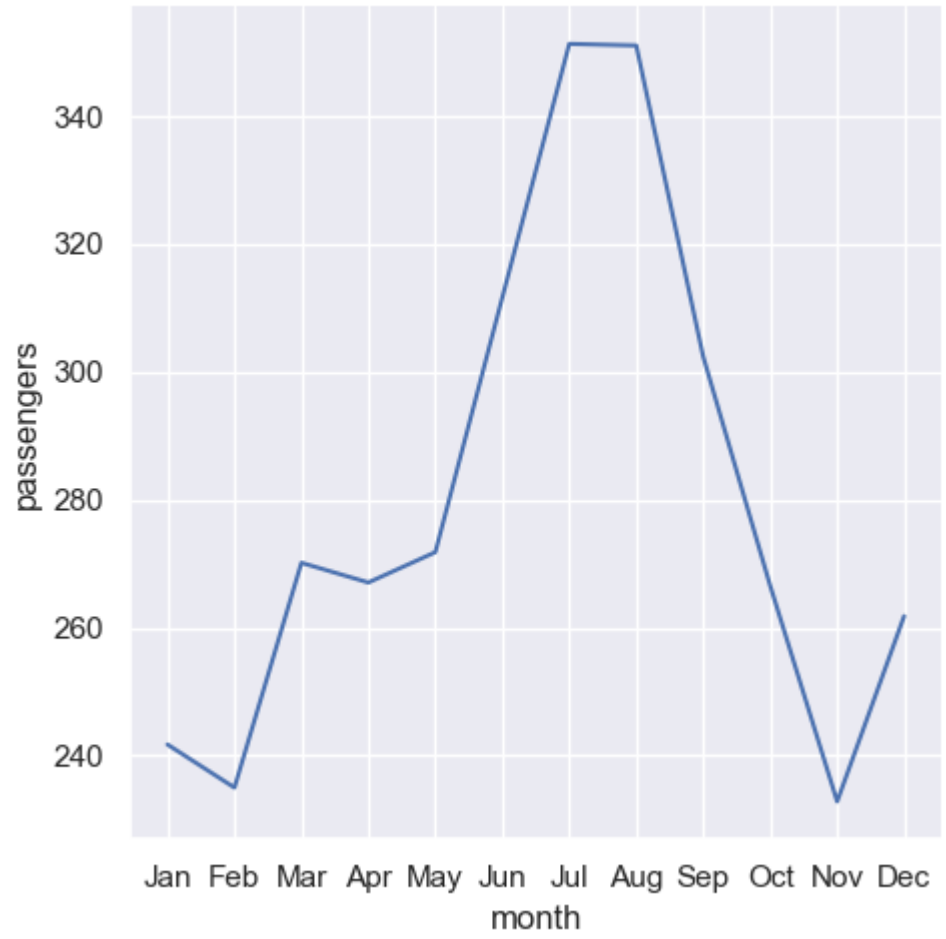
[7]:
```
sns.relplot(data=flights, x='year', y='passengers', kind='line');
```



We can remove the drawn error bars for the confidence intervals by setting the parameter `errorbar=None`. In the older versions of Seaborn this was controlled by the parameter `ci`. Therefore using `ci=None` still works, but it will produce a warning that `ci` has been deprecated.

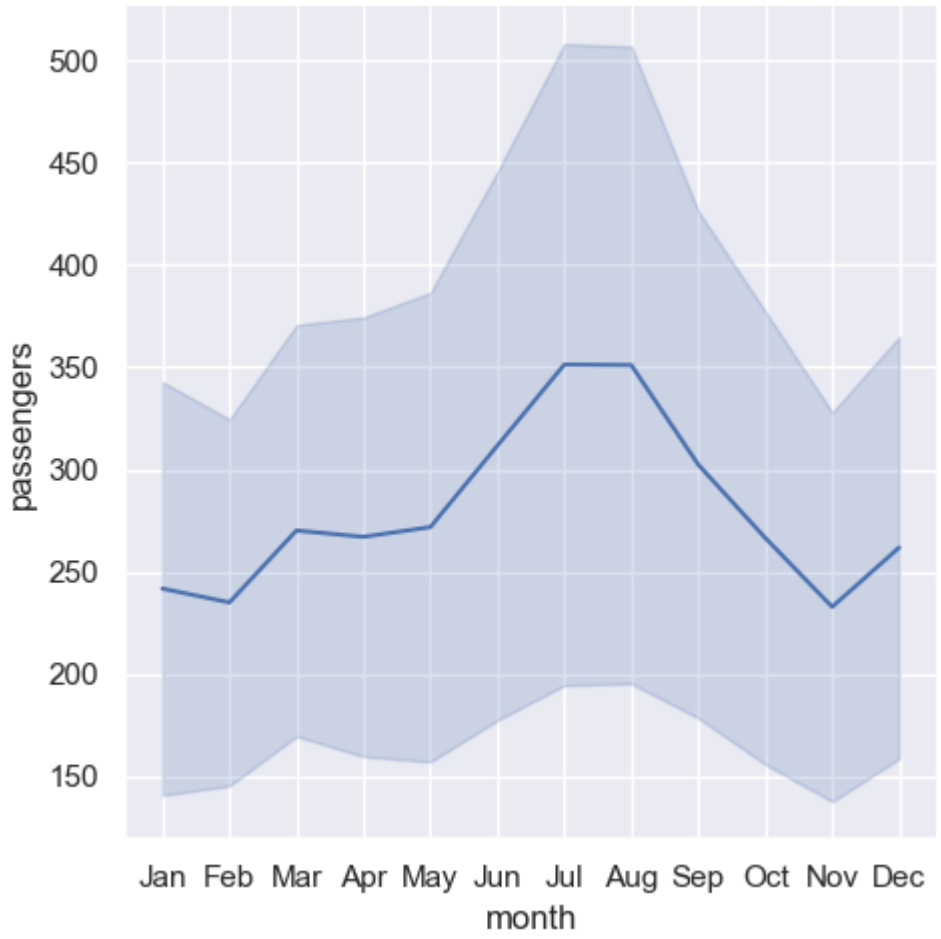
Similarly, by setting `style="darkgrid"` in `set_theme()` we indicate to set the background in the plot to a shade of dark, and to show the grid. By applying a theme with `set_theme()` we can control the appearance of all next plots.

[8]:
```
sns.set_theme(style="darkgrid") # show the grid for all next plots on a dark background
sns.relplot(data=flights, x='month', y='passengers', errorbar=None, kind='line');
```



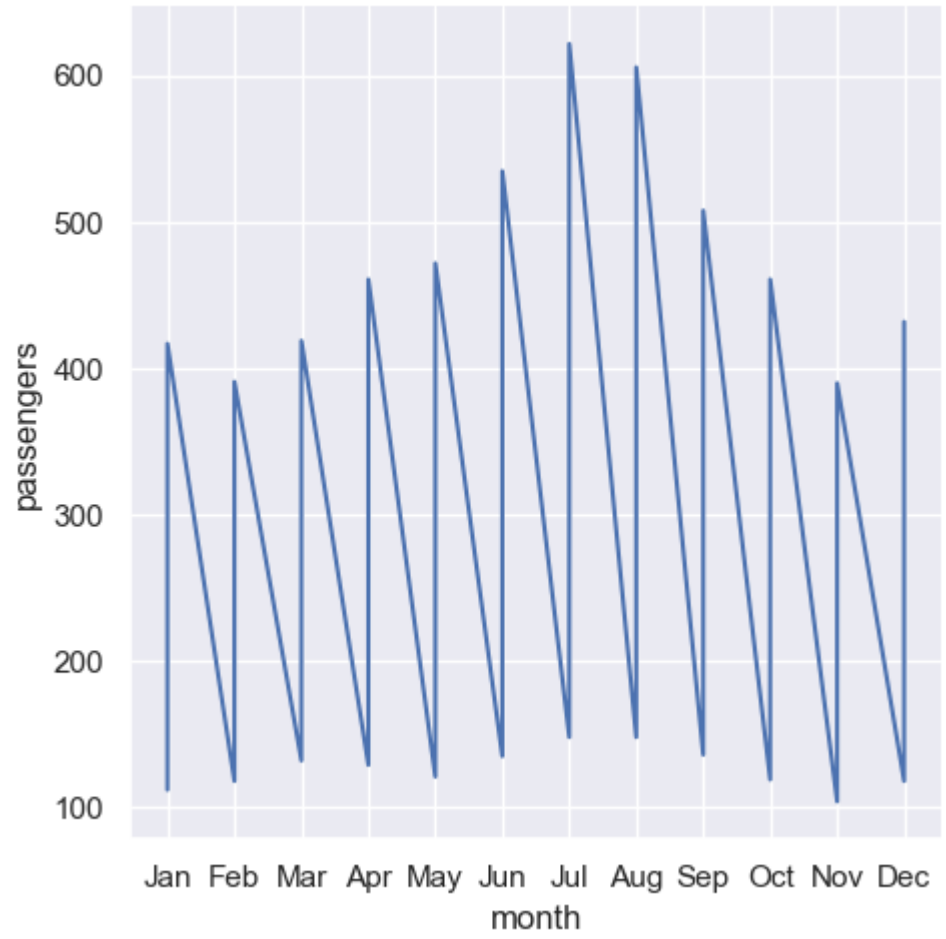
If we set the confidence interval to `errorbar=sd`, the standard deviation for the x-axis values will be shown in the plot.

[9]:
```
sns.relplot(data=flights, x='month', y='passengers', errorbar='sd', kind='line');
```



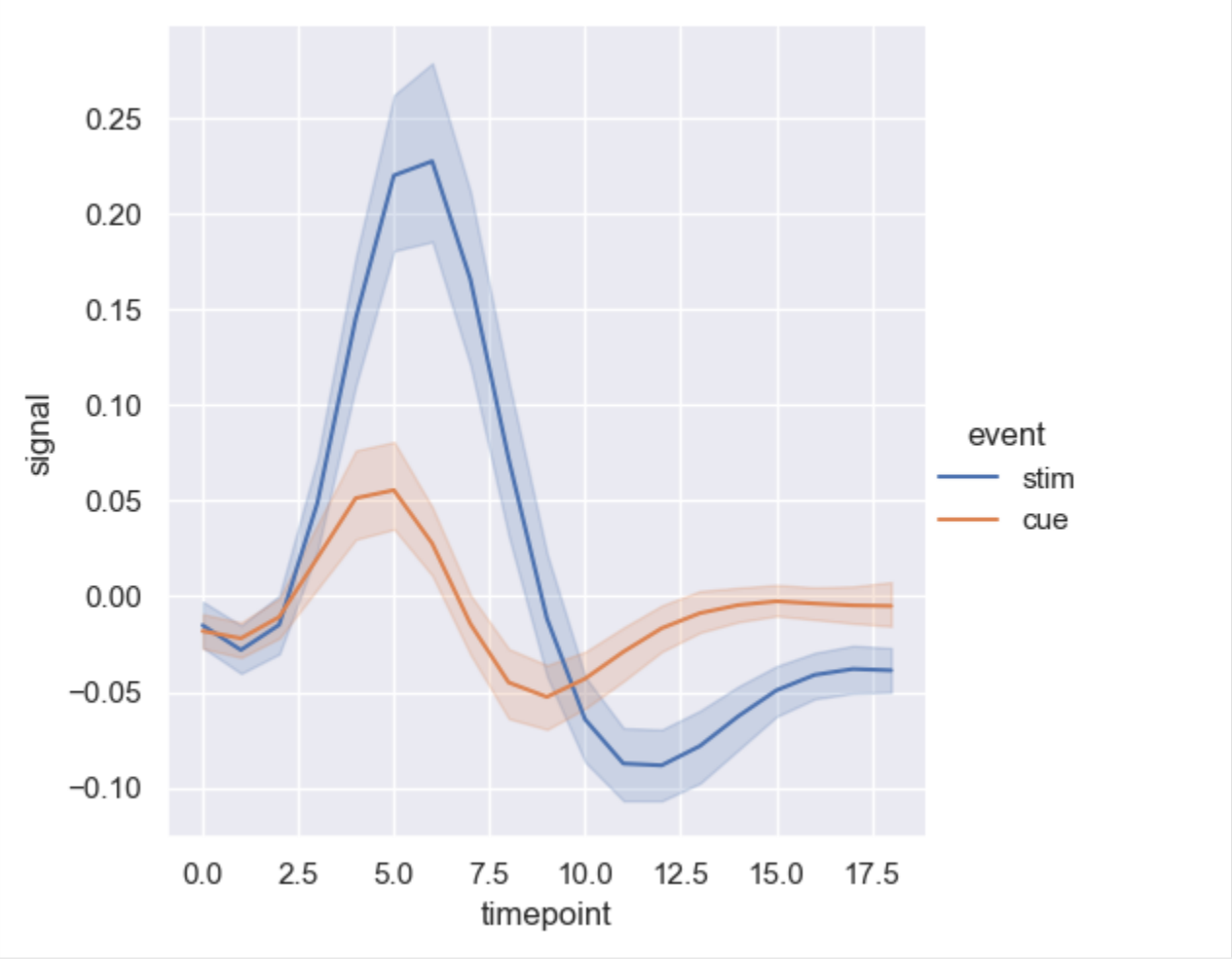
By default, `relplot()` employs a statistical estimator to aggregate the values and display the average values on the y-axis (e.g., average number of passengers per month). If we would like to see all values on the y-axis instead of the average values, we can set the parameter `estimator=None`.

[10]:
```
sns.relplot(data=flights, x='month', y='passengers', errorbar=None, estimator=None, kind=
↪'line');
```



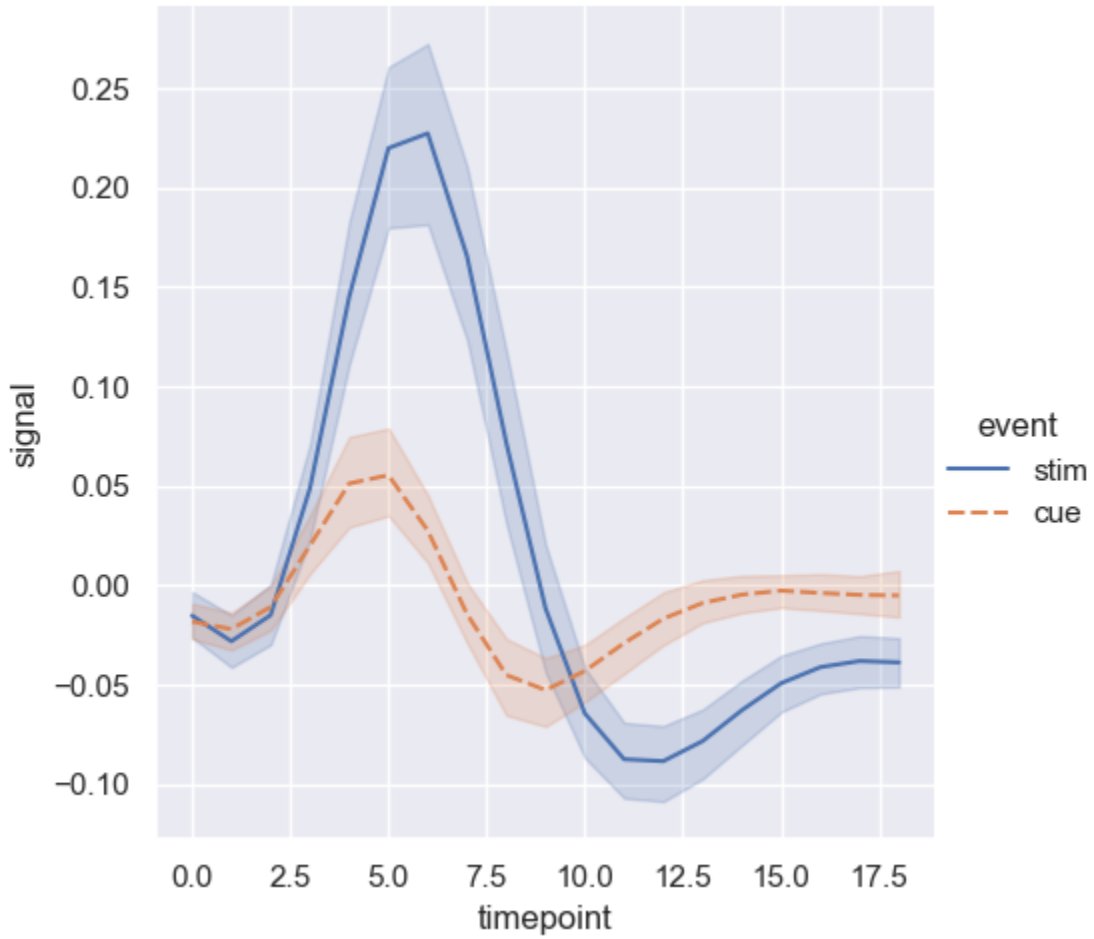
We can plot multiple lines in a figure by setting the `hue` parameter to another feature in the dataset.

[11]:
```
sns.relplot(data=fmri, x='timepoint', y='signal', hue='event', kind='line');
```



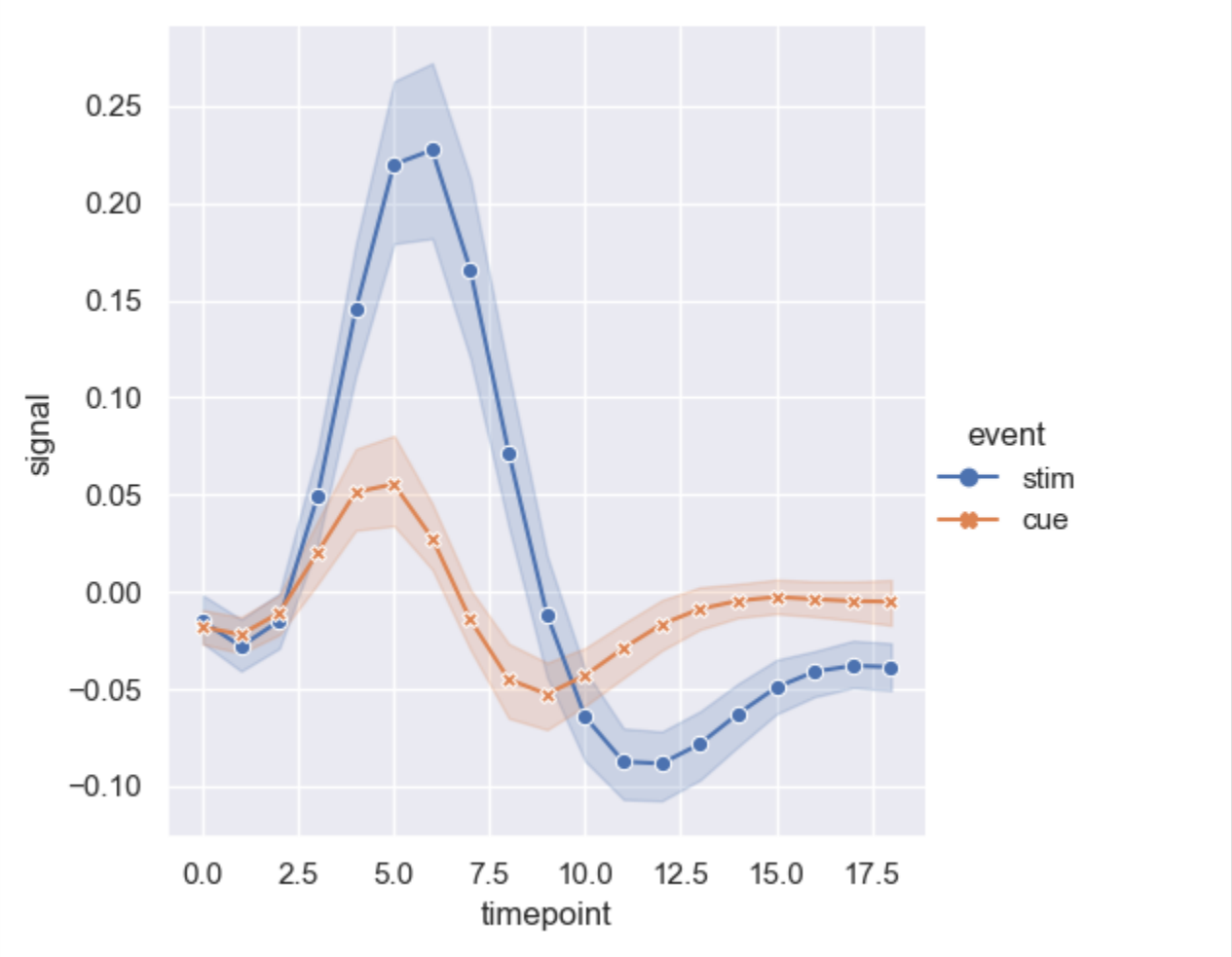
The parameter `style` allows to vary the appearance of the lines based on a specified feature (in the next cell, based on the feature `event`). Seaborn will automatically assign a different line style (e.g., dashed line) to each category in the feature `event`. In this case, there are two categories: `stim` and `cue`.

[12]:
```
sns.relplot(data=fmri, x='timepoint', y='signal', hue='event', kind='line', style='event
↪');
```



To highlight the differences between the categories in `hue`, we can use `markers=True` to add markers for the data points.

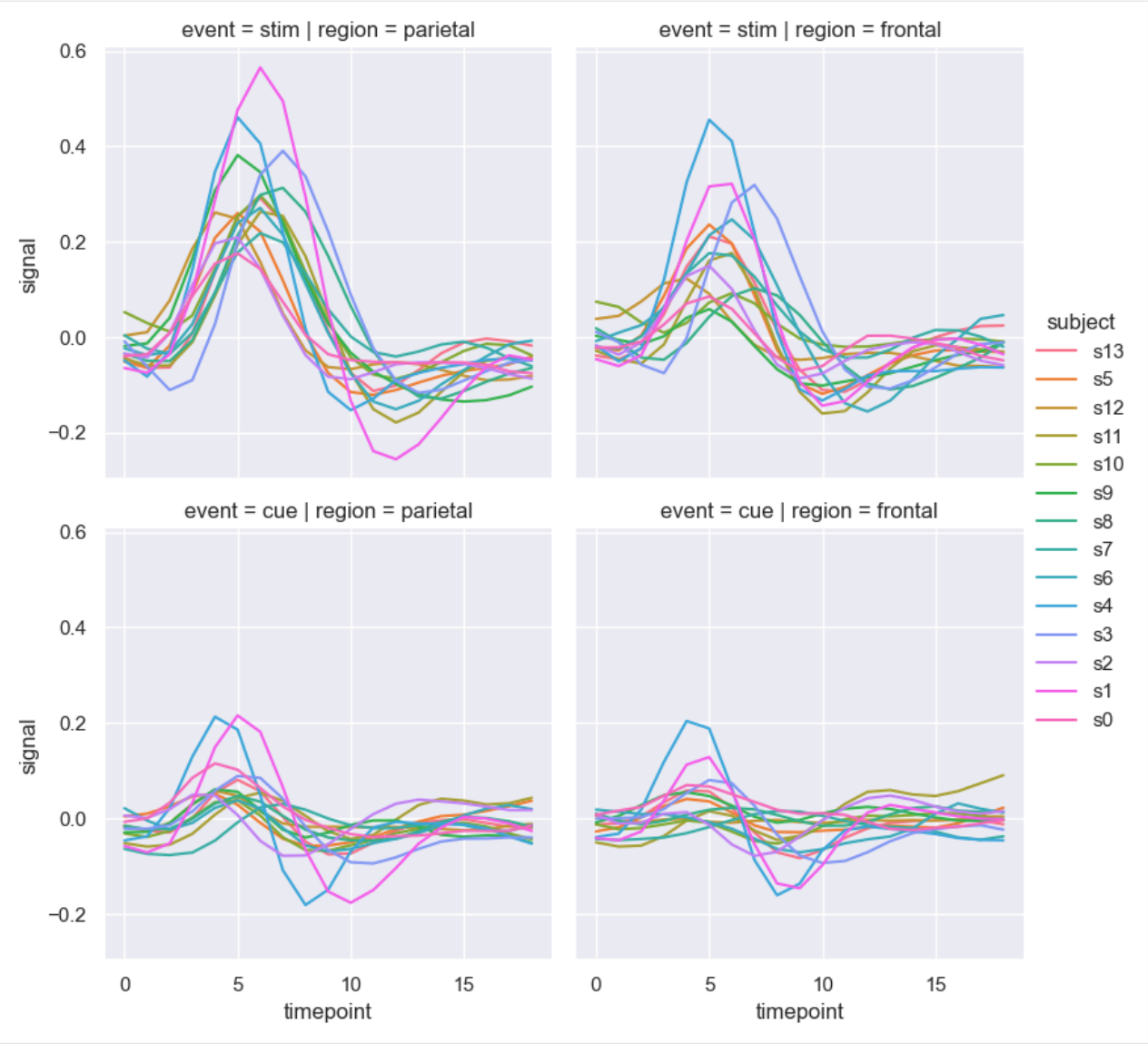
[13]:
```
sns.relplot(data=fmri, x='timepoint', y='signal', hue='event', kind='line',
            style='event', dashes=False, markers=True);
```



And, we can plot multiple relationships by introducing a parameter `col` to create multiple columns, or we can also introduce multiple rows with the `row` parameter, as in the following cell below.

[14]:
```
sns.relplot(data=tips, x='total_bill', y='tip', hue='smoker', col='time');
```

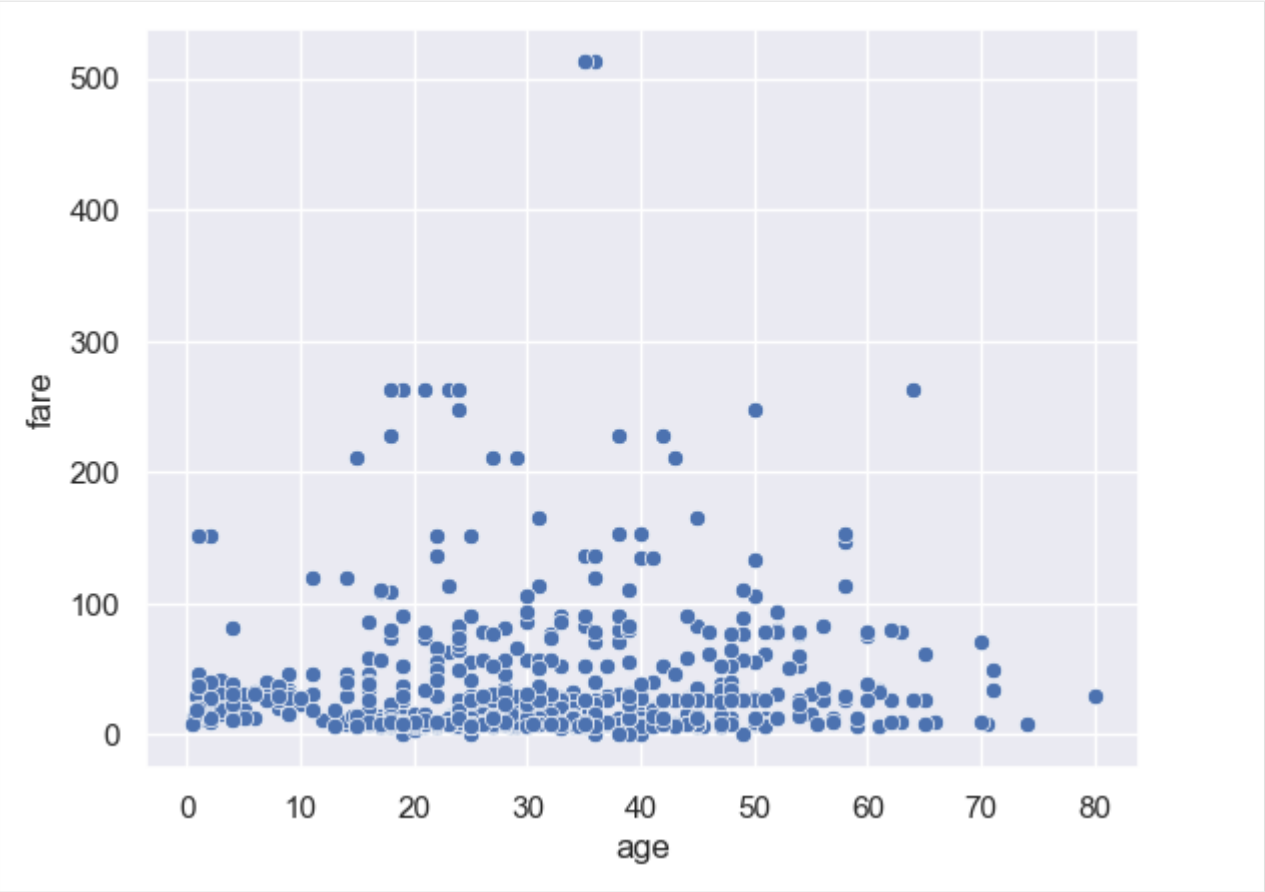
[15]:
```
sns.relplot(data=fmri, x='timepoint', y='signal', hue='subject',
            col='region', row='event', height=4, kind='line', estimator=None);
```



Scatter Plots

We can create scatter plots in Seaborn by using either `sns.scatterplot()` or `sns.relplot()`.

[16]:
```
sns.scatterplot(data=titanic, x='age', y='fare');
```



Similar to line plots, the hue parameter distinguishes the data points in the plot based on another feature.

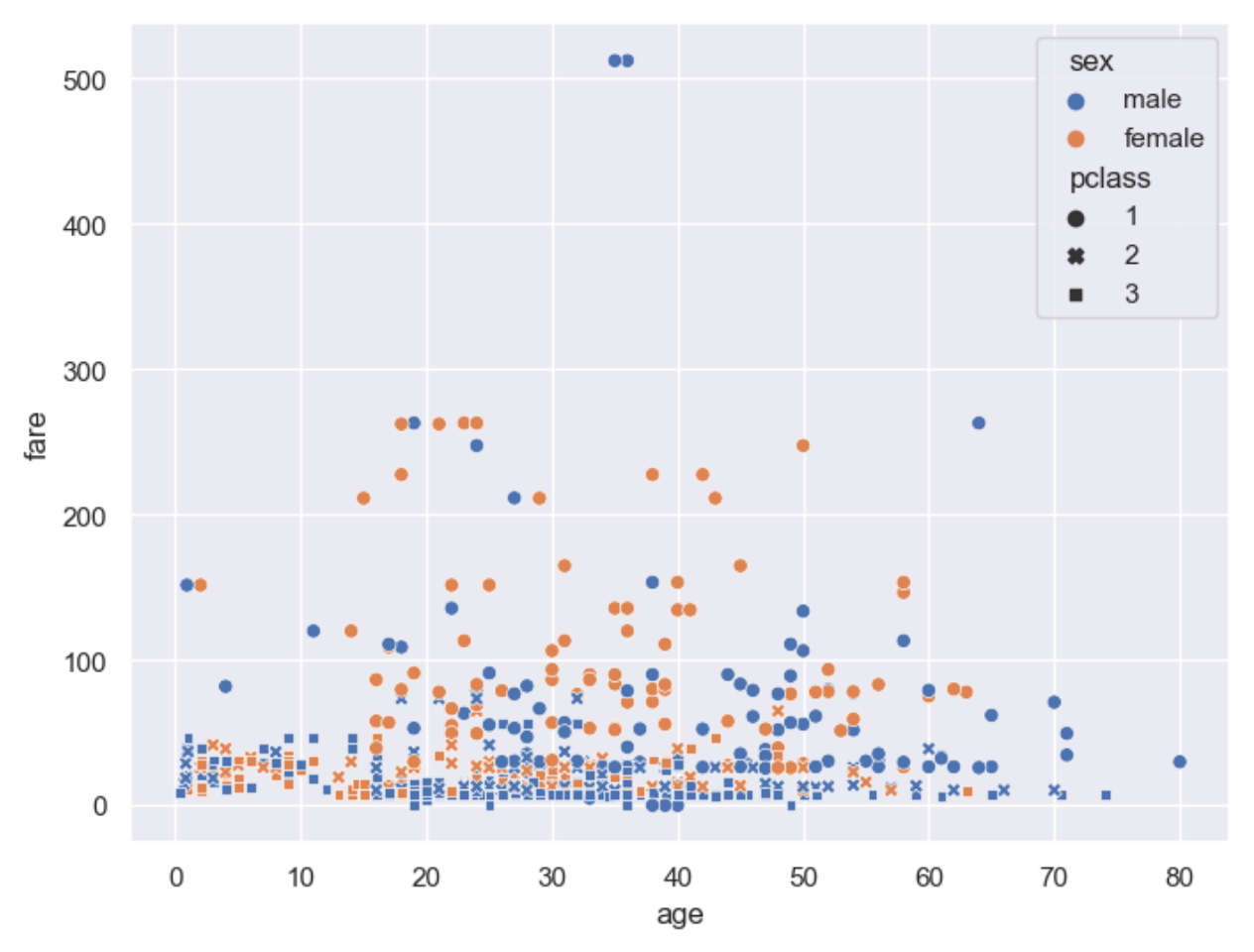
[17]:
```
sns.scatterplot(data=titanic, x='age', y='fare', hue='sex');
```

To highlight the difference between the categories in `hue`, we can add different marker styles with `style='sex'`.

[18]:
```
sns.scatterplot(data=titanic, x='age', y='fare', hue='sex', style='sex');
```

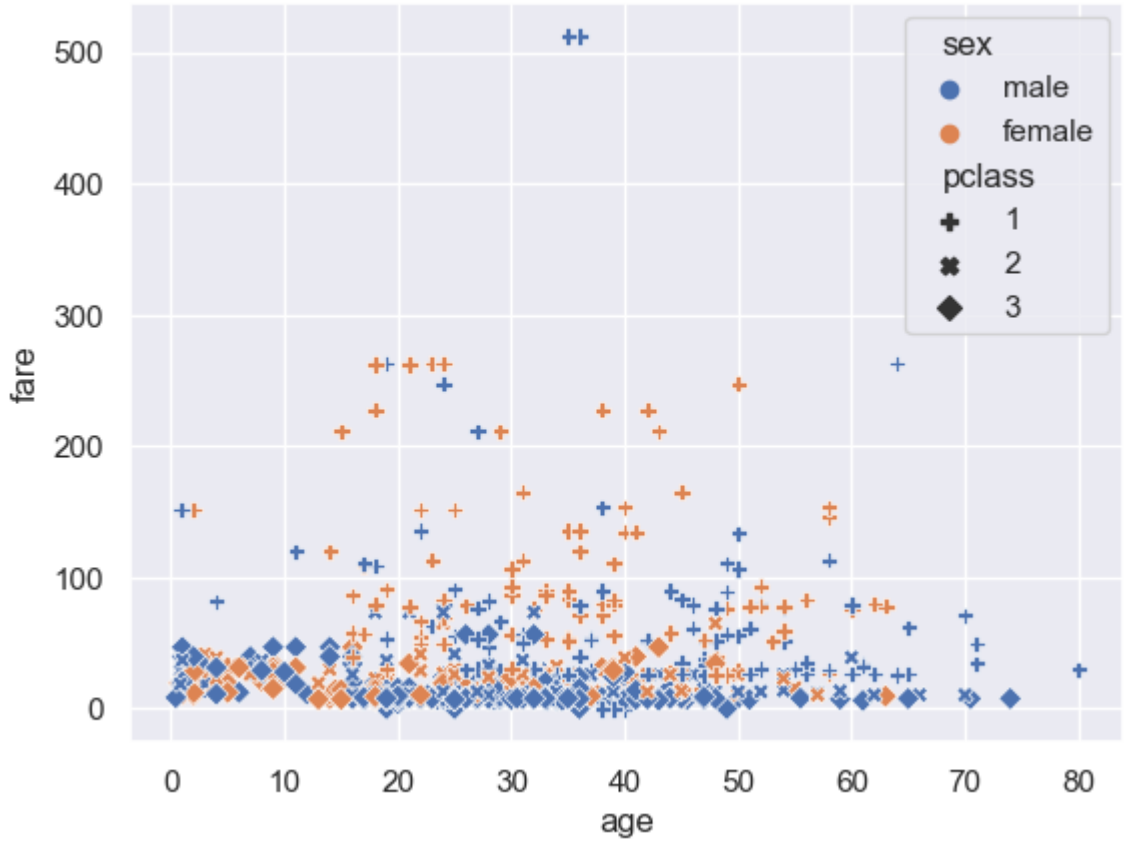
Recall that the default size of figures in Matplotlib is 6.4 by 4.8 inches. The same applies to Seaborn, and similar to Matplblib, to specify the figure size, we can use `figure(figsize=())`.

[19]:
```
plt.figure(figsize=(8,6))
sns.scatterplot(data=titanic, x='age', y='fare', hue='sex', style='pclass');
```



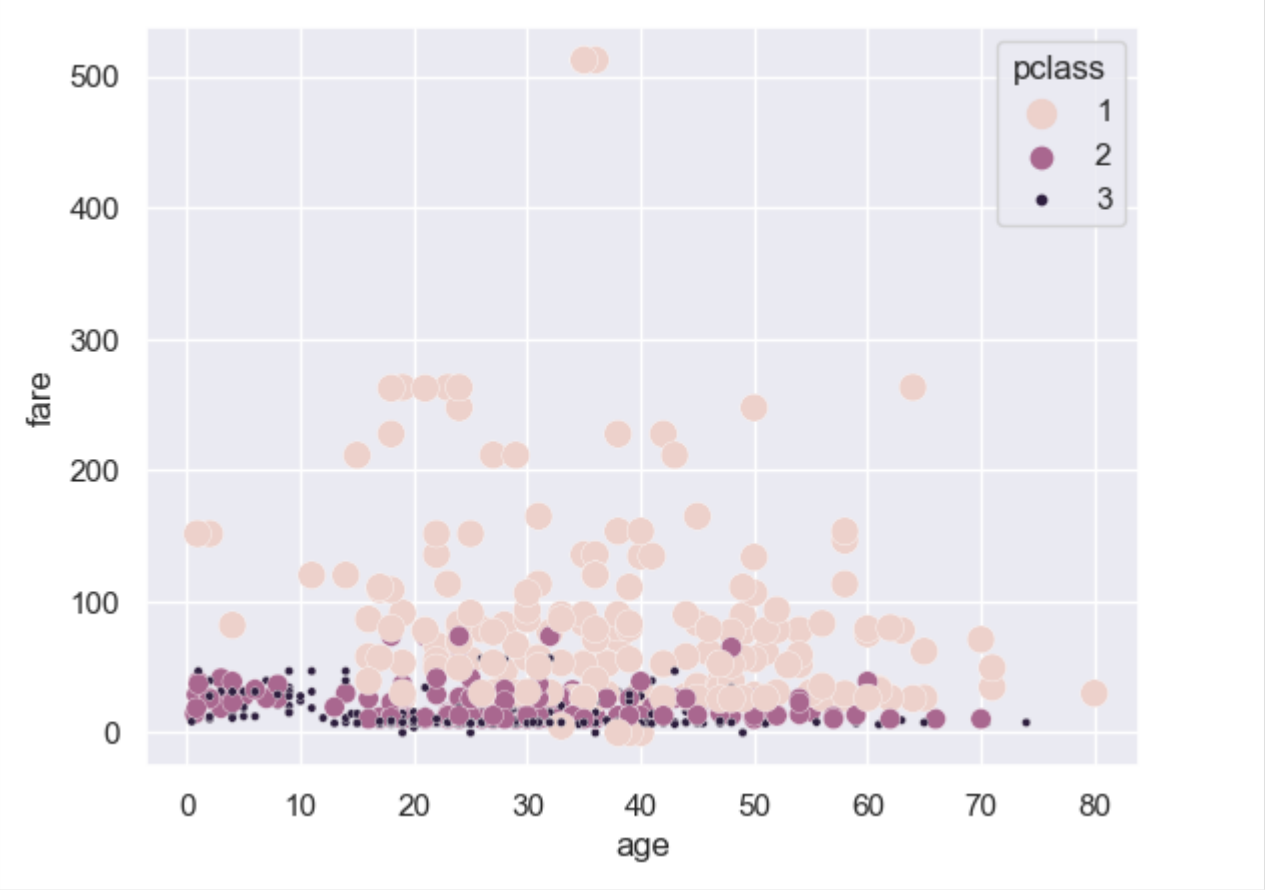
We can also define the type of markers to use with the parameter `markers`.

[20]:
```
marker_types = {1:'P', 2:'X', 3:'D'} # P(plus), X (cross), and D (diamond) markers
sns.scatterplot(data=titanic, x='age', y='fare', hue='sex', style='pclass',␣
↪markers=marker_types);
```



Adding the parameter `sizes` can make the plot more meaningful, as Seaborn will use `sizes` to control the size of the markers in the plot.

[21]:
```
sns.scatterplot(data=titanic, x='age', y='fare', hue='pclass', size='pclass', sizes=(100,
↪10));
```



7.12.2 12.2 Distributional Plots

Distributional plots visualize the distribution of data features, and can help understand the range of values, their central tendency, potential skewness in the data, presence of outliers, and other data characteristics.

Distribution plot functions in Seaborn include:

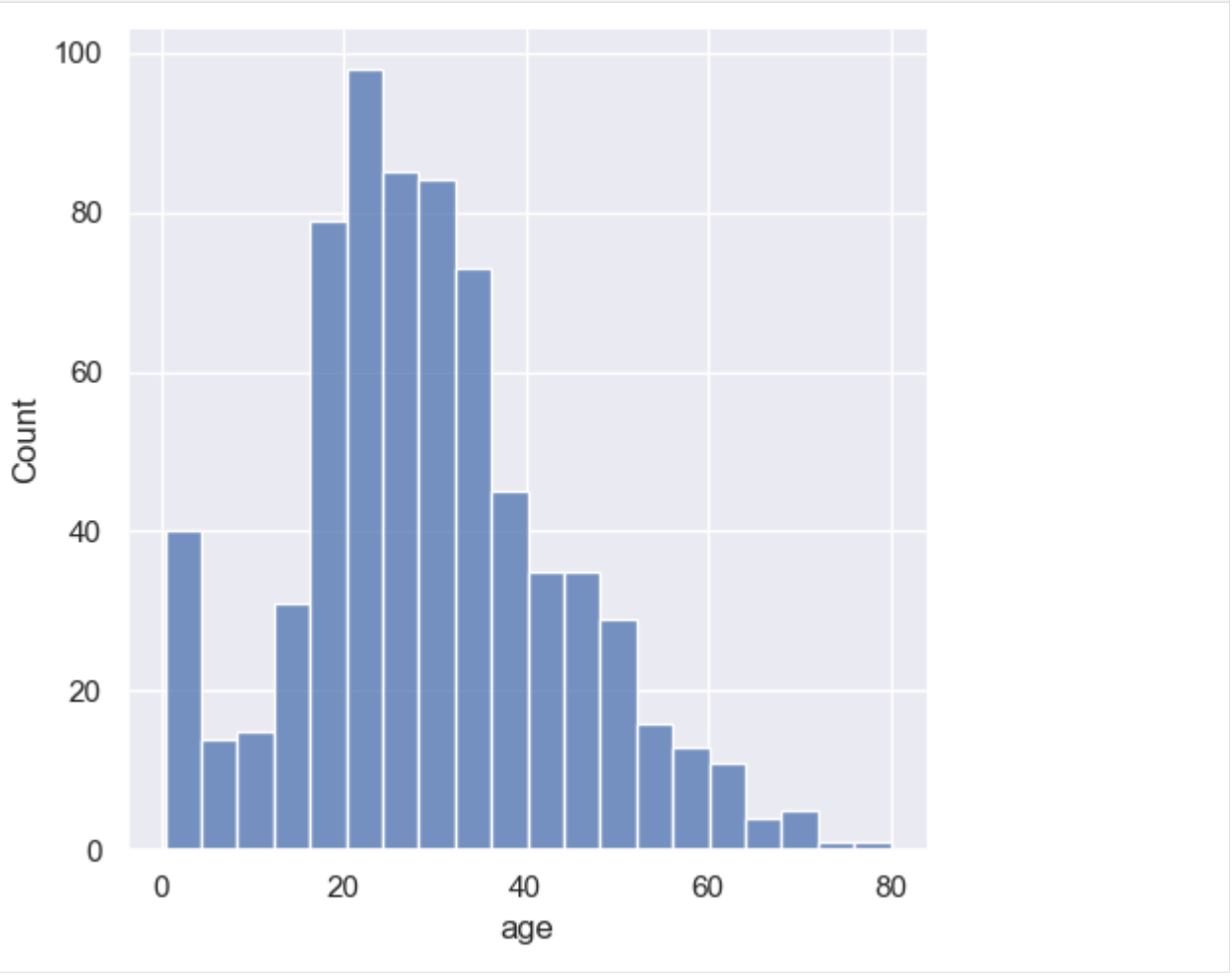
- `displot()`
- `histplot()`
- `jointplot()`
- `pairplot()`
- `rugplot()`
- `kdeplot`

Univariate Analysis with `displot()`

Univariate analysis explores the characteristics of a single feature in a data set in isolation. It involves studying the range of values, central tendency of the values, and other statistical properties of the feature. This analysis typically employs histograms, bar plots, box plots, and other related plots for presenting the distribution of an individual feature.

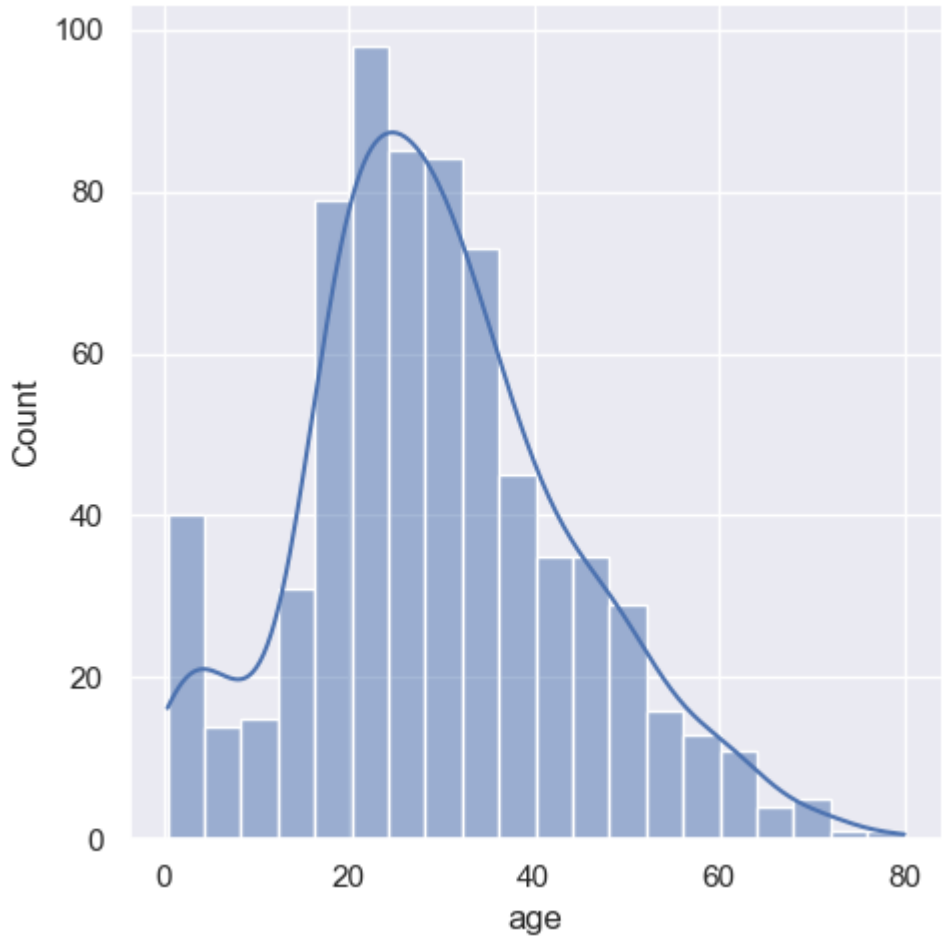
In Seaborn, the `displot()` function is a general function for plotting different distributional plots. Its default behavior is to draw a histogram, and internally `displot()` uses the `histplot()` function to draw histograms. By changing the parameter `kind` in `displot()`, we can create different distributional plots.

[22]:
```
sns.displot(titanic['age']);
```



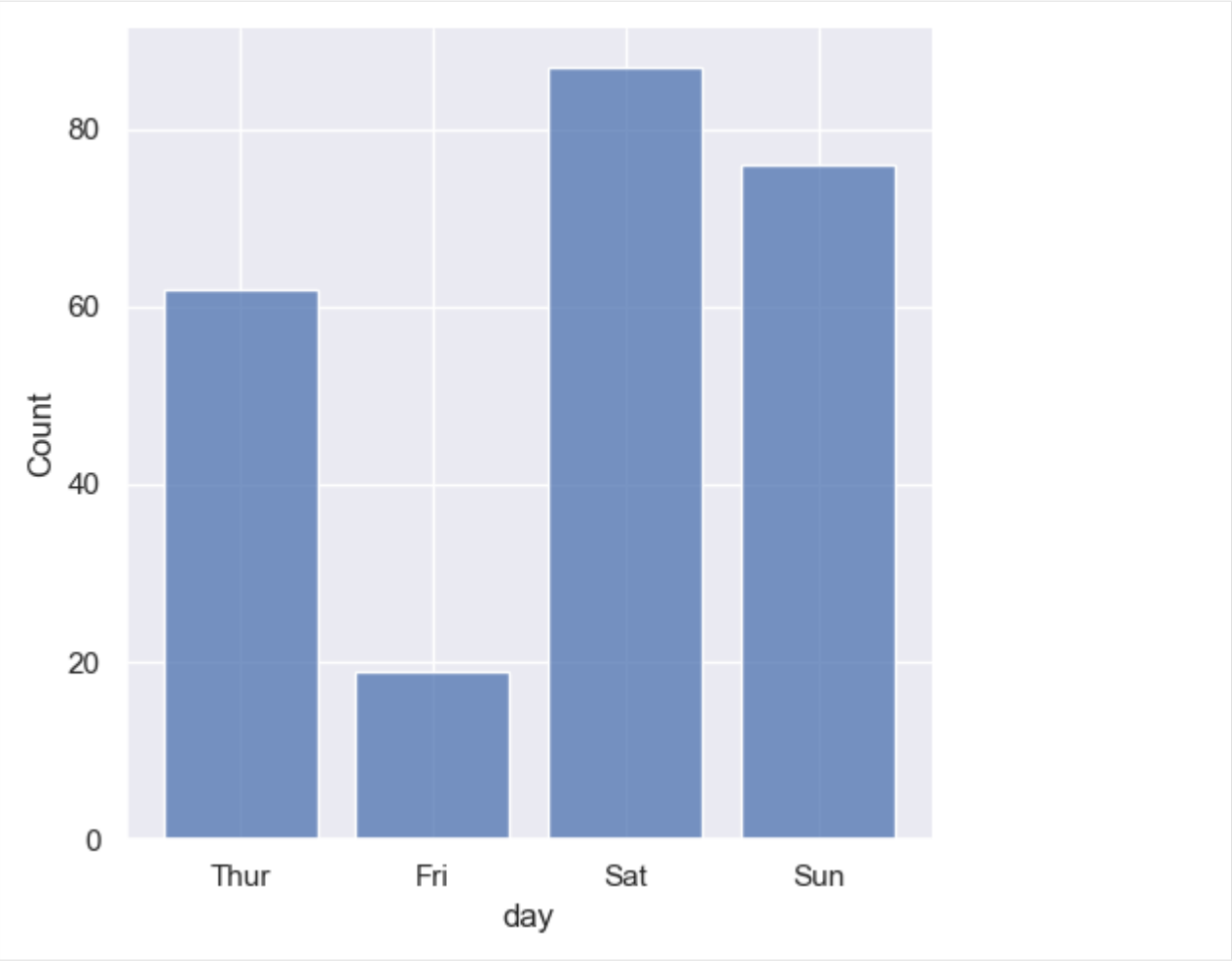
We can set the number of `bins` in histograms to a value of choice. Similarly, if we set the parameter `kde` which stands for Kernel Density Estimator (KDE) to True, a KDE plot will be overlaid on top of the histogram. KDE is a smoothed curve representation of the probability density function of the data. It uses a Gaussian kernel to smooth the data distribution.

[23]:
```
sns.displot(titanic['age'], kde=True, bins=20);
```

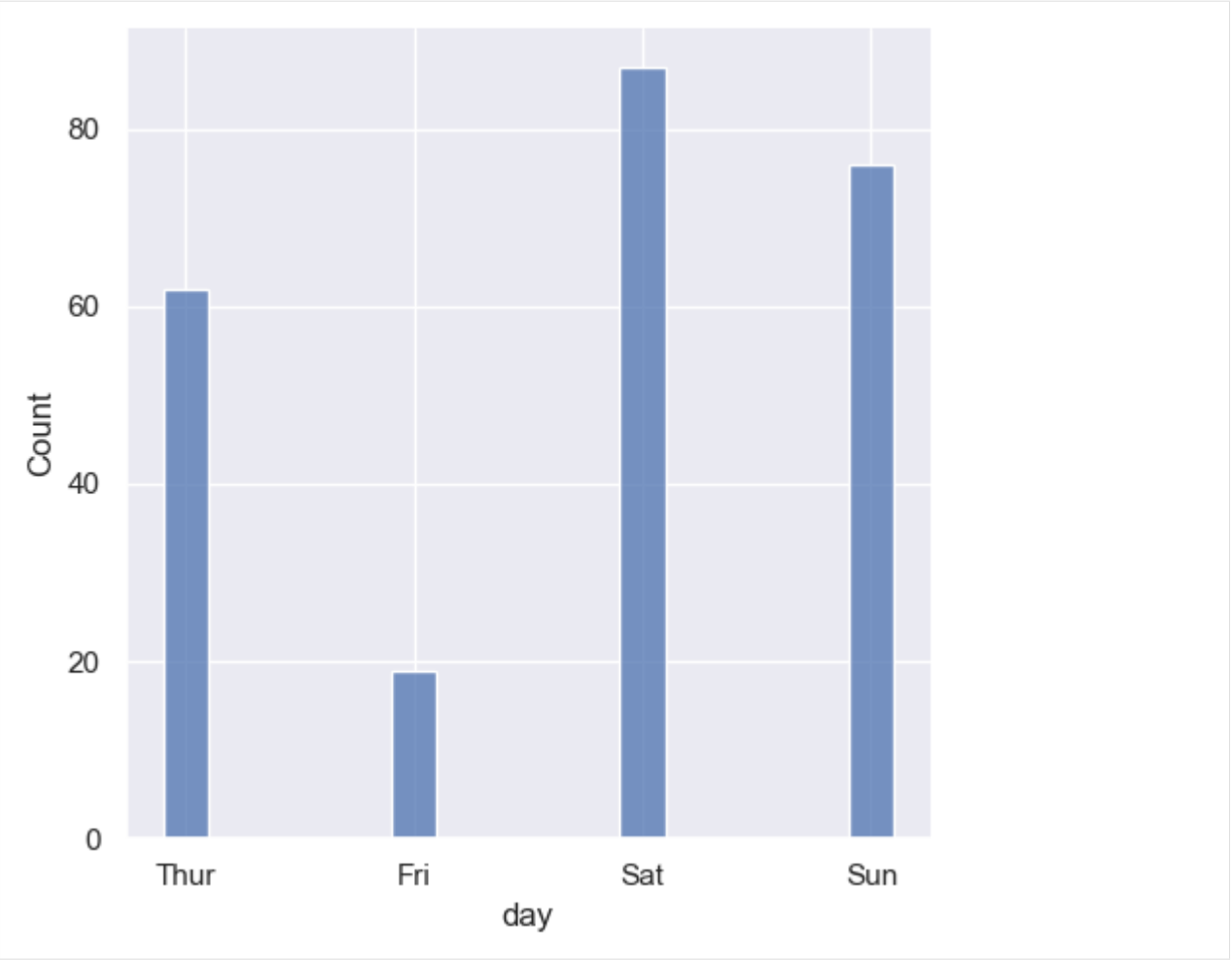


We can also plot categorical data with histograms, in which case the histograms become bar plots. The parameter `shrink` in the next cells has a value between 0 and 1 and controls the width of the bars in the plot. Compare the following two cells.

[24]:
```
sns.displot(tips, x='day', shrink=0.8);
```

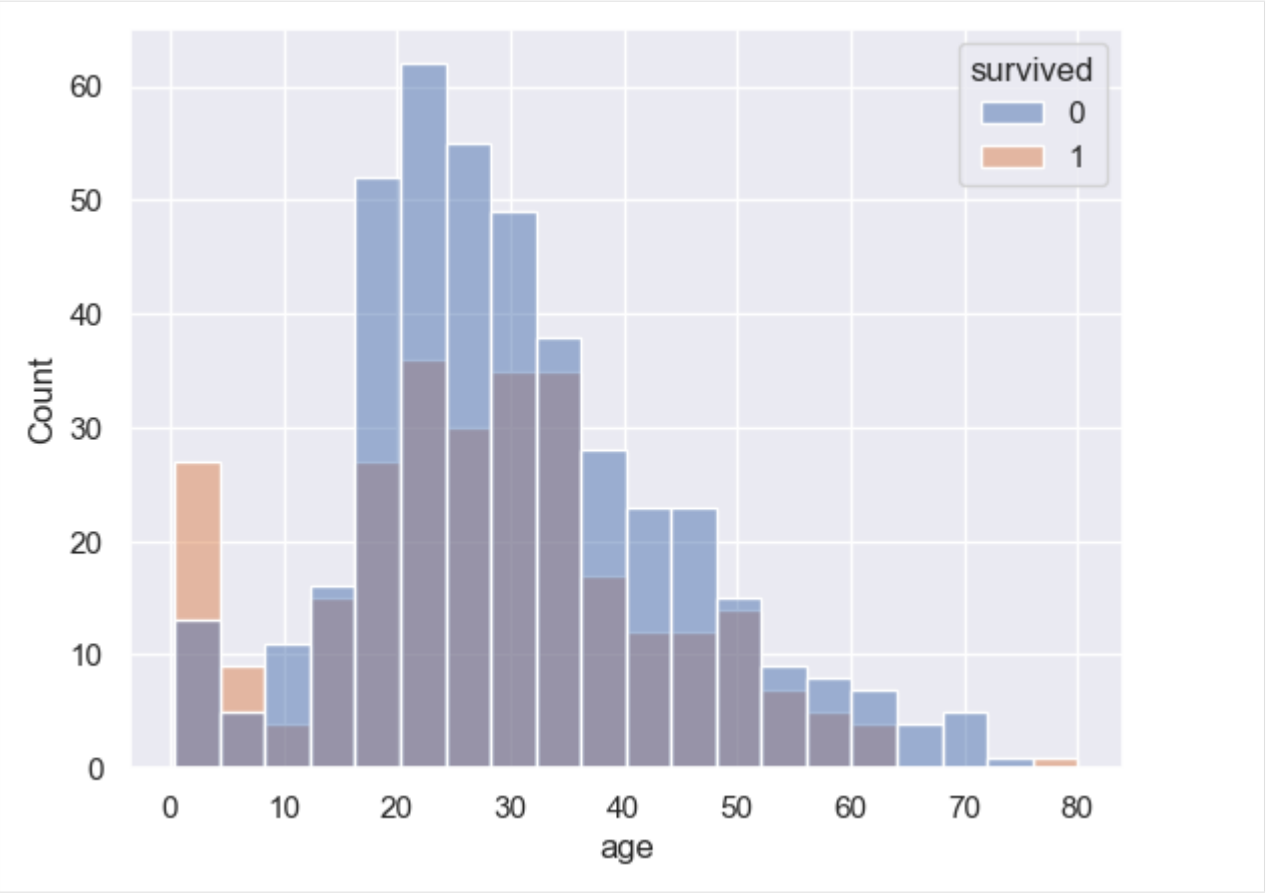


[25]:
```
sns.displot(tips, x='day', shrink=0.2);
```



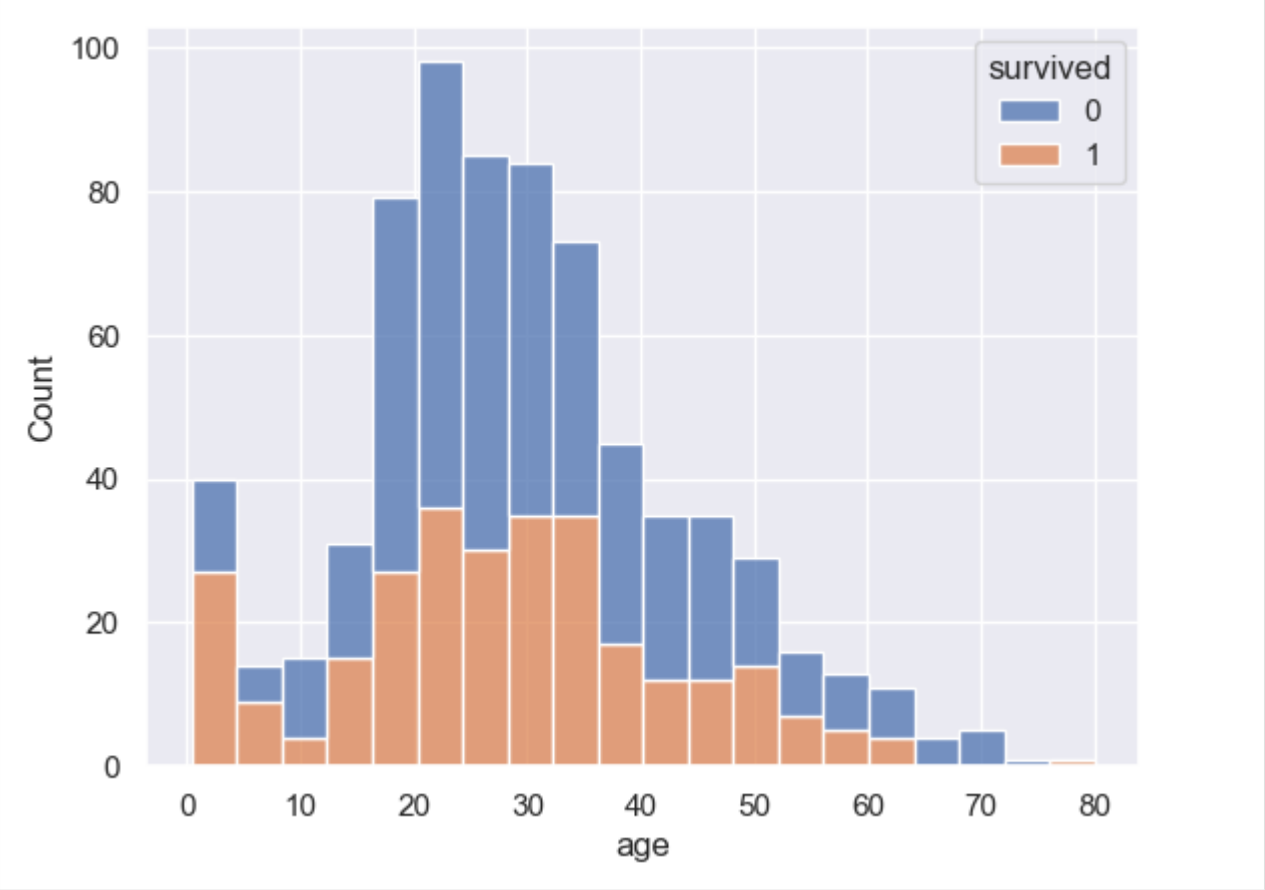
We can also use `histplot()` to draw histograms.

[26]:
```
sns.histplot(titanic, x="age", hue="survived");
```



We can stack histograms of multiple features by setting the parameter `multiple` to `'stack'`.

[27]:
```
sns.histplot(titanic, x='age', hue='survived', multiple='stack');
```



Bivariate Analysis with `jointplot()`

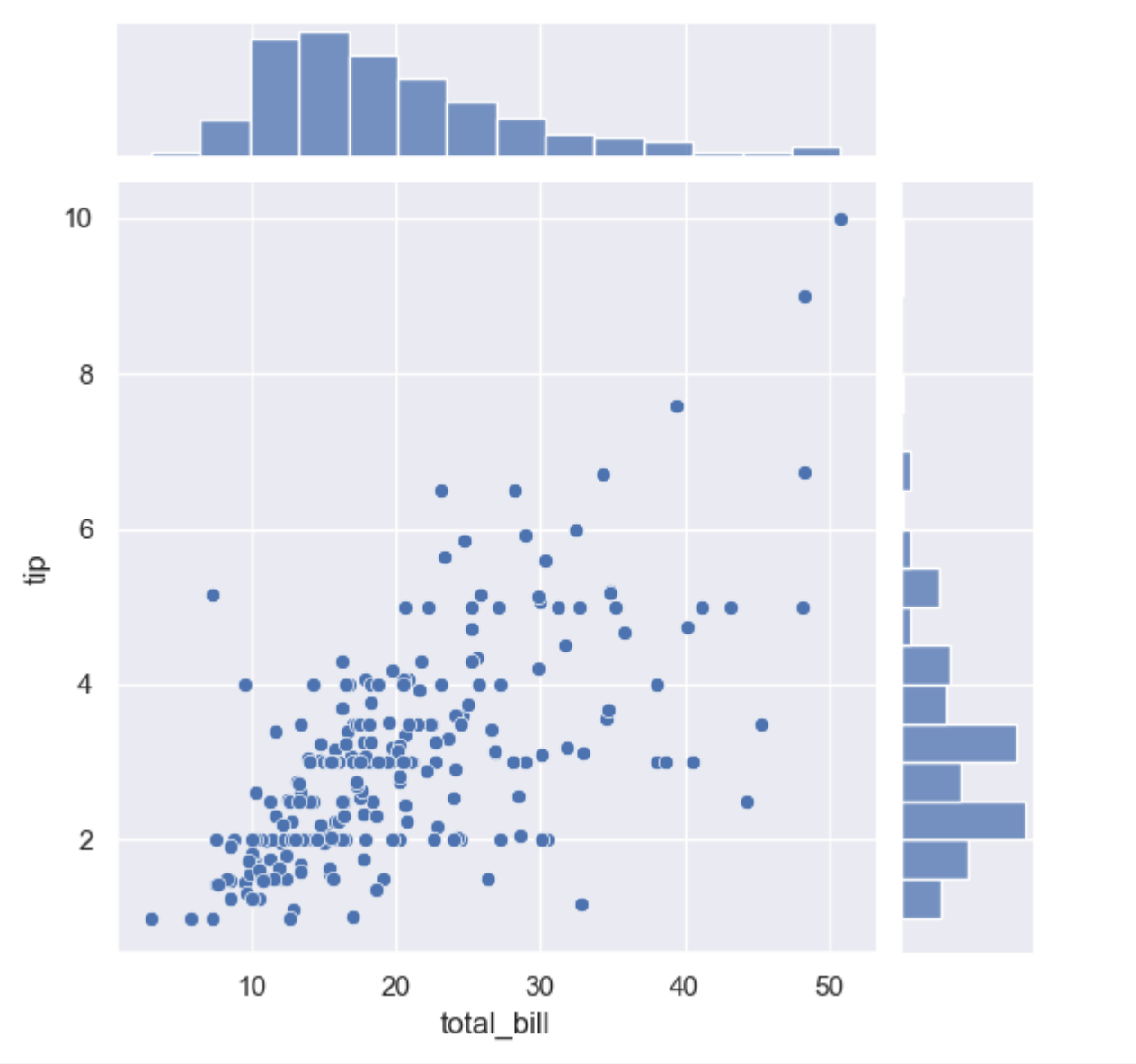
Bivariate analysis explores the relationship and patterns among two variables, or two features in a dataset. The goal is to understand how changes in one variable are related to changes in another variable. Common plots in bivariate analysis include scatter plots, and correlation plots.

In Seaborn, we can use `jointplot()` to plot the joint distribution between two features along with the marginal distribution for each of the features. This function creates multi-panel plots, and it provides useful statistical visualization for any two features in a dataset.

The **joint distribution** drawn in the central plot displays the simultaneous behavior of the two features as they both change, i.e., it provides information about the probability of different combinations of values for the two features.

The **marginal distribution** plots are shown on the side axes, and they present the distribution of each feature when considered independently by ignoring or "marginalizing out" the other feature. I.e., they show the histogram for numerical features or the bar plot for categorical features.

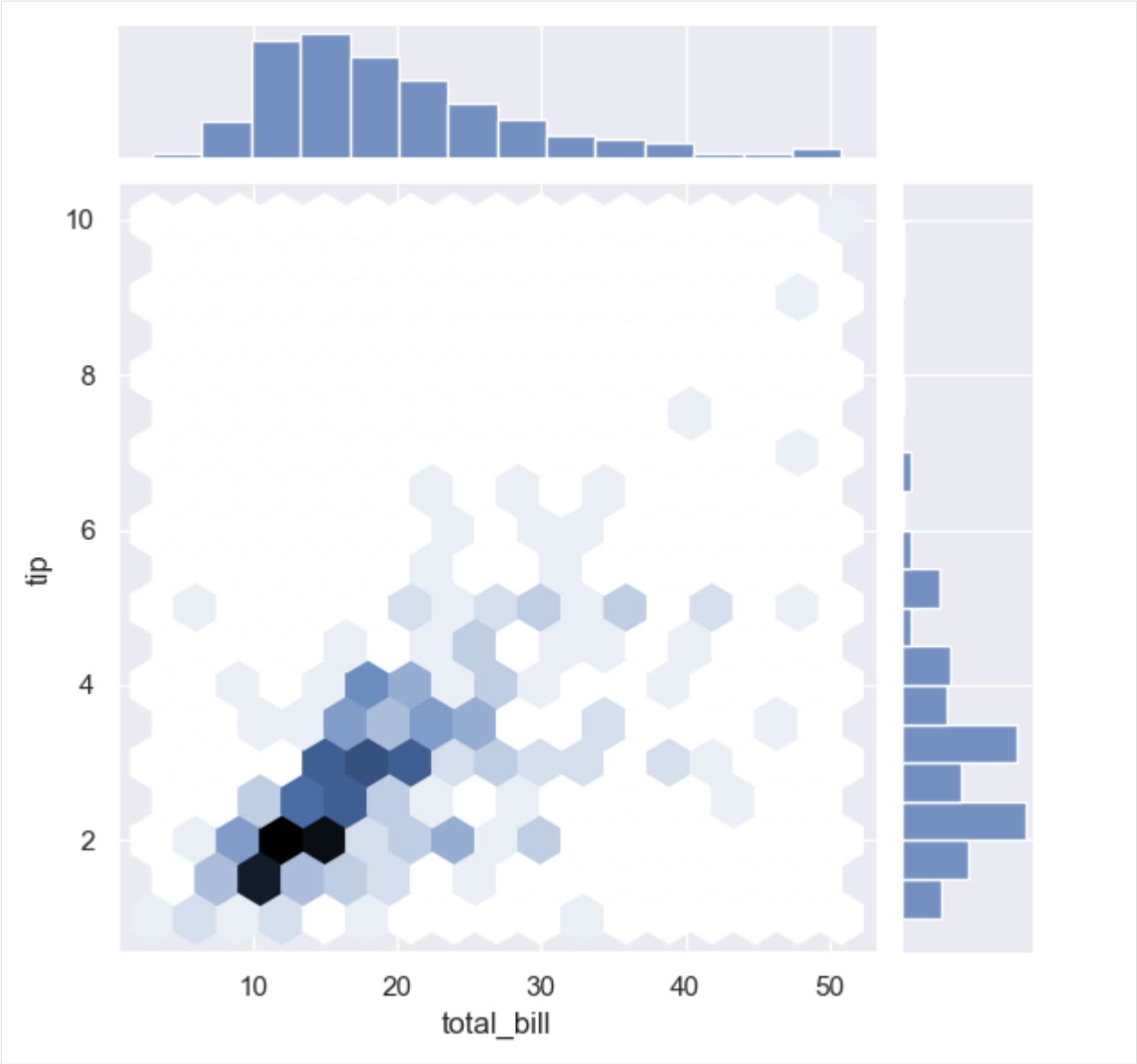
[28]:
```
sns.jointplot(data=tips, x='total_bill', y='tip');
```



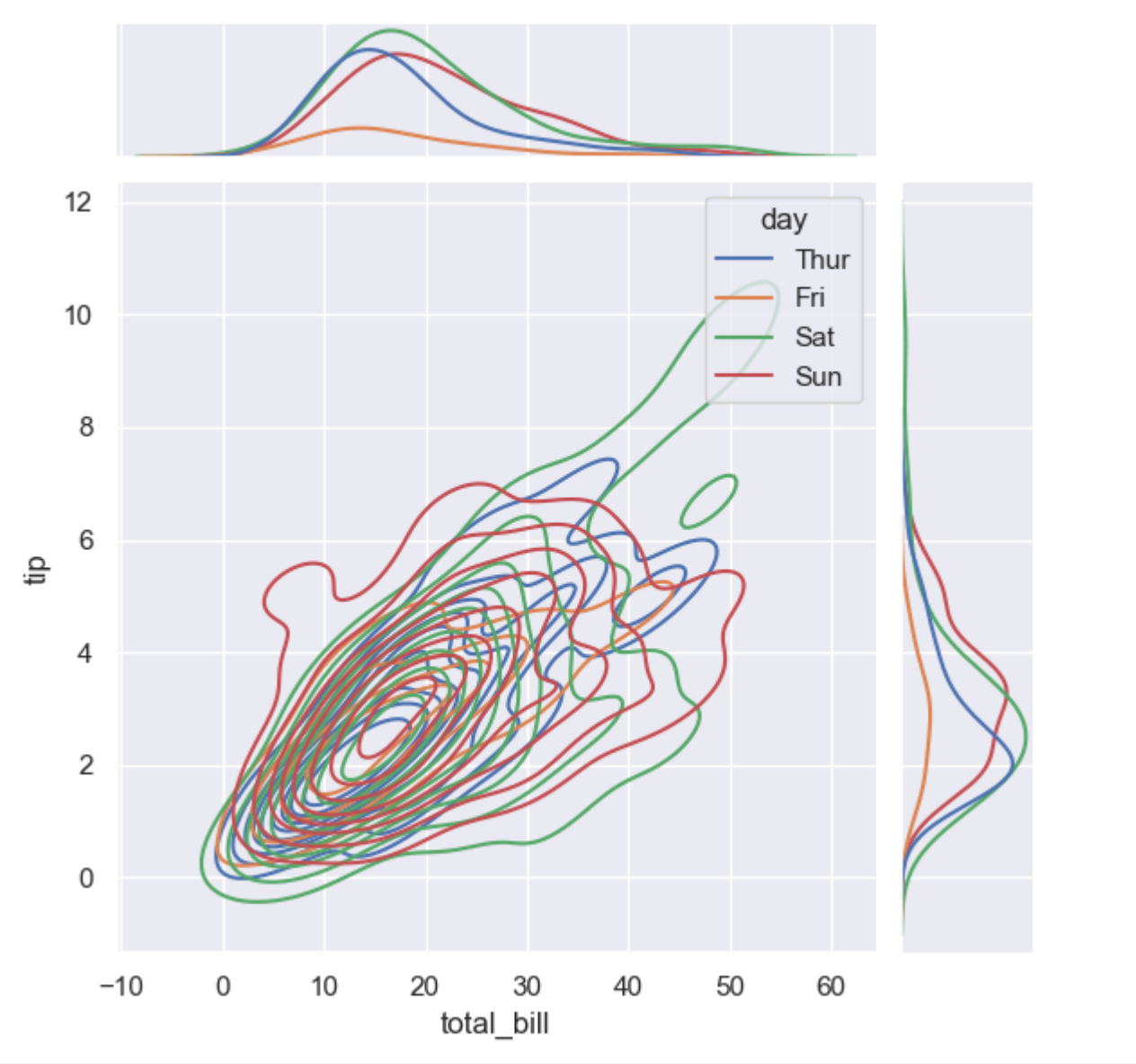
In the function `jointplot()`, the following options for the parameter `kind` are available:

- `scatter` (default) displays a scatter plot
- `hex` displays a hexagonal binning style to visualize the density
- `kde` displays kernel density estimates
- `reg` displays a scatter plot with a linear regression line
- `resid` displays the residuals of a linear regression, useful for checking goodness of fit

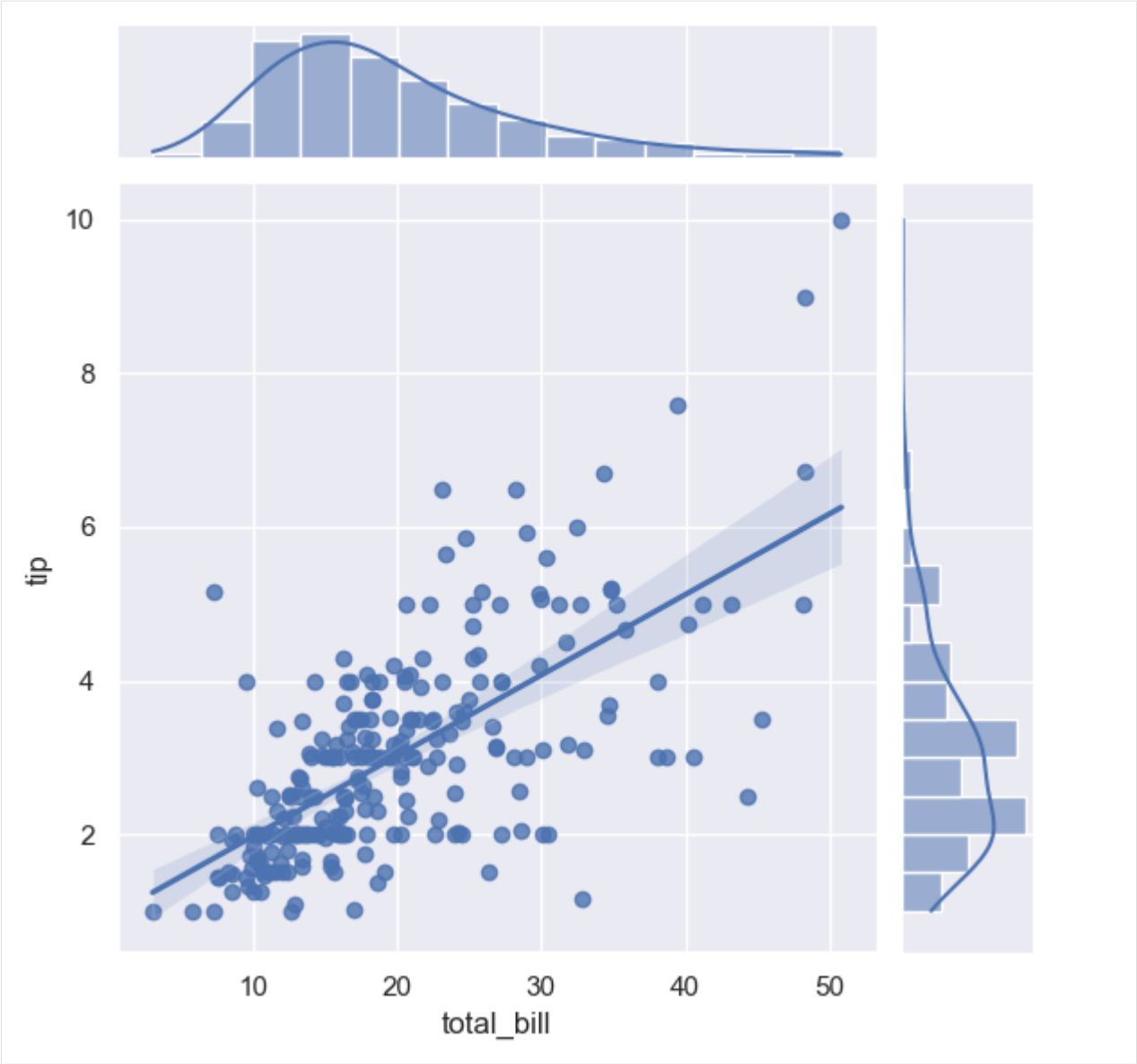
[29]:
```
sns.jointplot(data=tips, x='total_bill', y='tip', kind='hex');
```



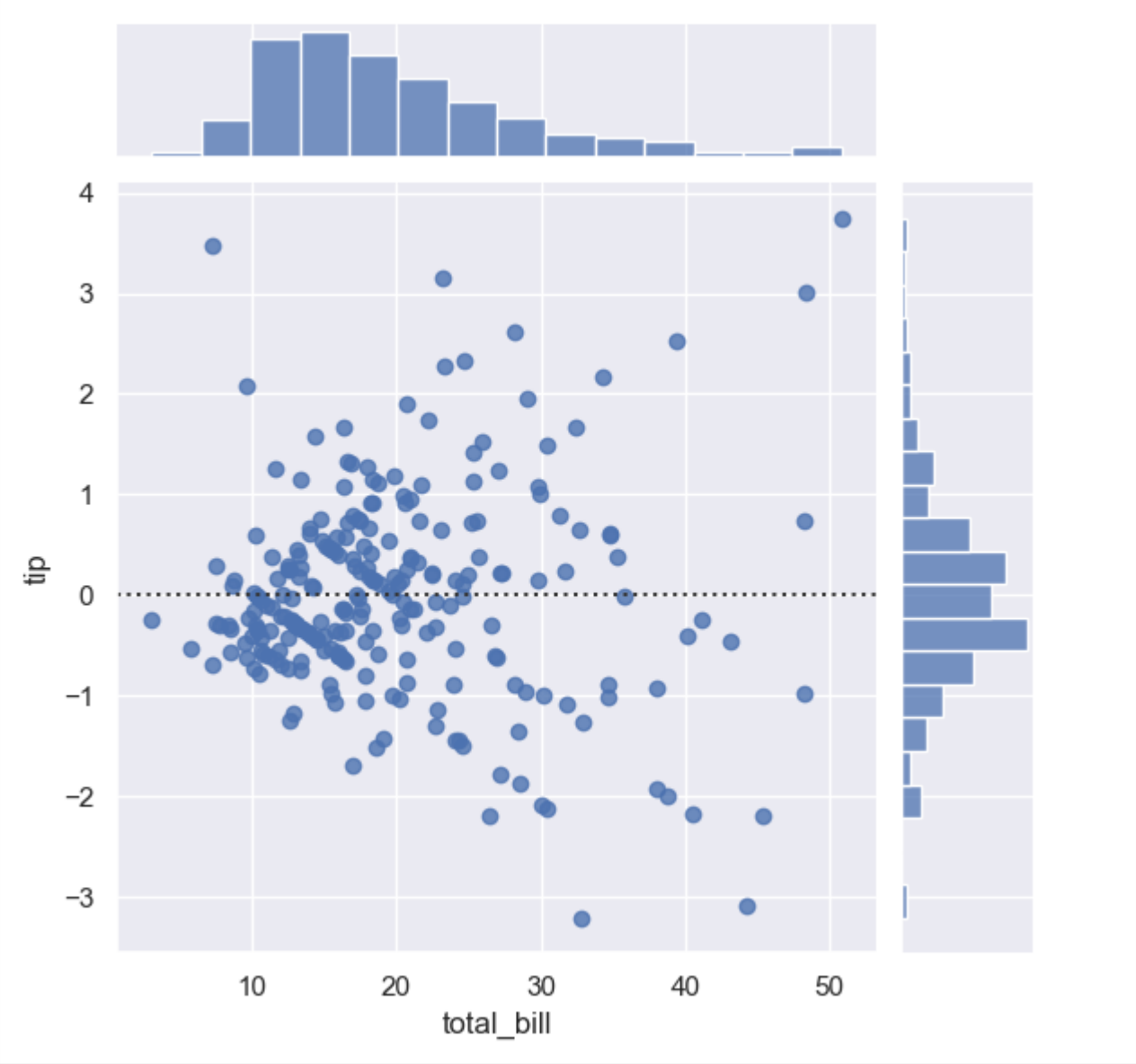
[30]:
```
sns.jointplot(data=tips, x='total_bill', y='tip', hue='day', kind='kde');
```



[31]:
```
sns.jointplot(data=tips, x='total_bill', y='tip', kind='reg');
```



[32]:
```
sns.jointplot(data=tips, x='total_bill', y='tip', kind='resid');
```



Multivariate Analysis with `pairplot()`

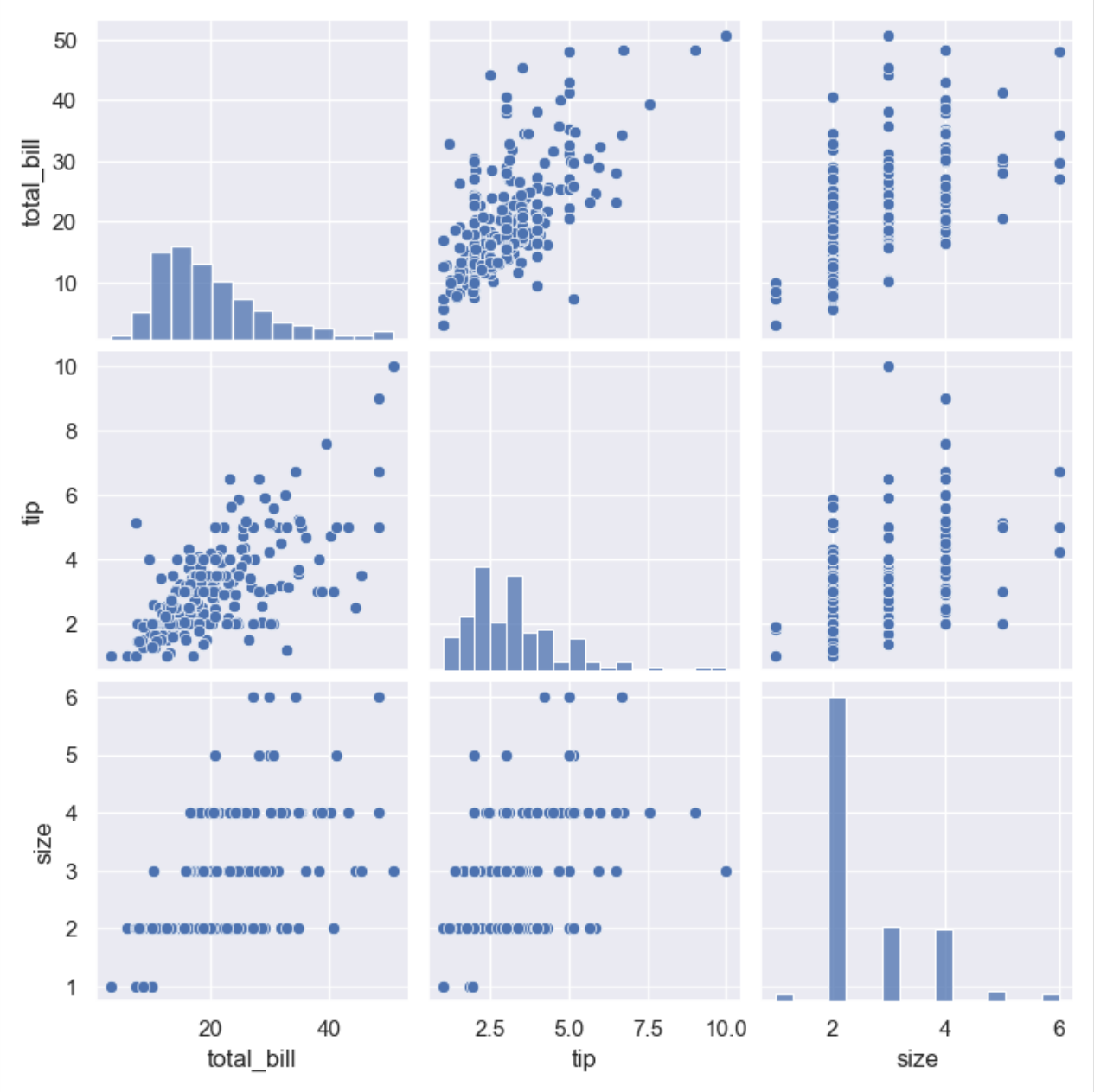
Multivariate analysis explores relationships and dependencies among multiple features in a dataset, in order to reveal complex interactions between more several features.

The function `pairplot()` visualizes all possible joint and marginal distributions for all pairwise relationships and for each variable in datasets. It allows to immediately notice relationships between the features.

In particular, `pairplot()` creates a grid of subplots, where the diagonal subplots are the marginal distributions of each feature (i.e., histograms for numerical features and bar plots for categorical features), and the off-diagonal subplots draw the pairwise joint distributions between different pairs of features.

Note the difference: while `jointplot()` visualizes the relationship between two features (bivariate analysis), `pairplot()` visualizes the pairwise relationships between all features in a dataset (muliivariate analysis).

[33]:
```
sns.pairplot(tips);
```

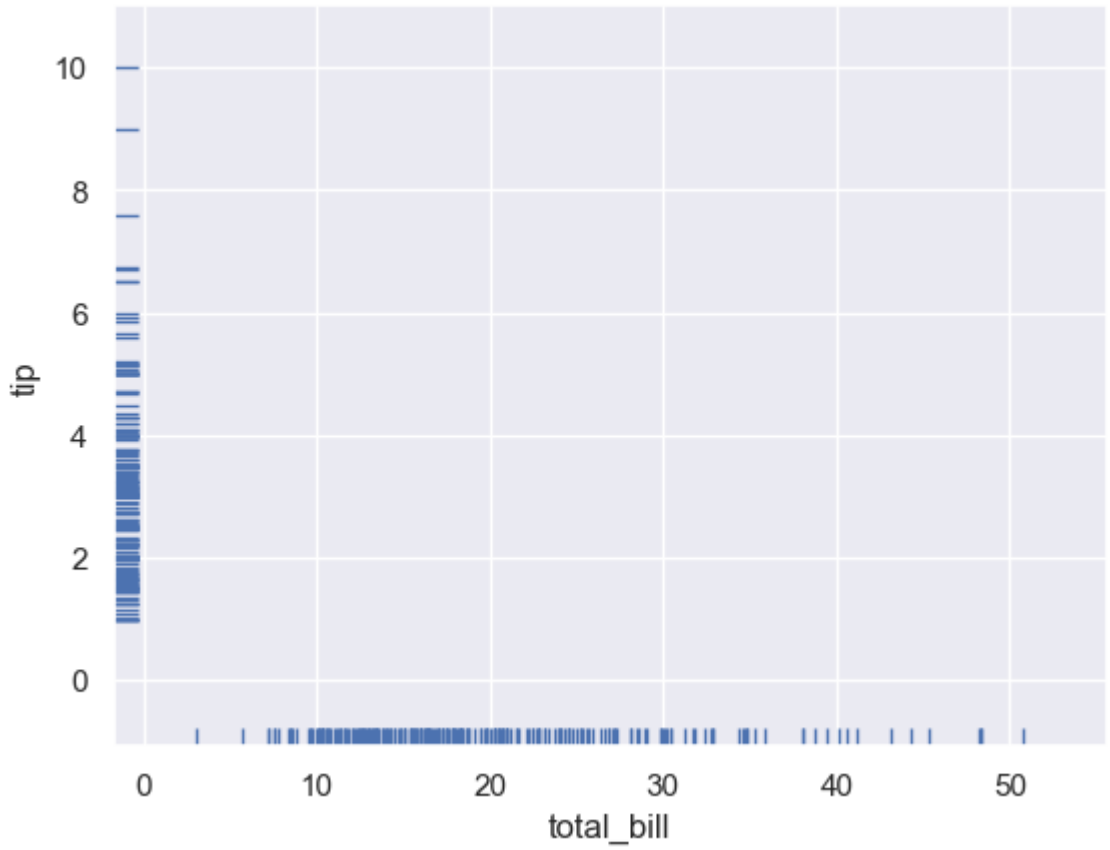


Plotting Distributions with `rugplot()`

The function `rugplot()` visualizes marginal distributions by drawing ticks or dashes along the x-axis and/or y-axis in a plot. It consists of short vertical 'ticks' where each tick shows the presence of a data point at that value on the axis. This plot is useful for showing the density of data points along an axis.

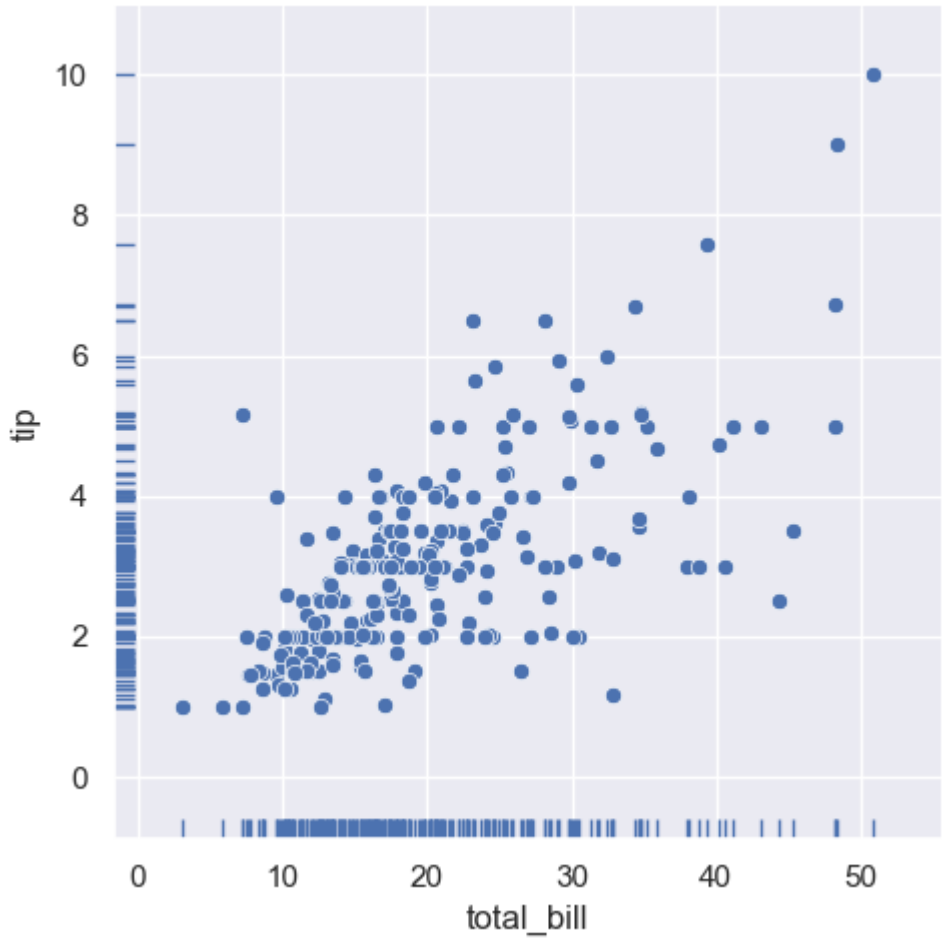
The ticks in the plot resemble the appearance of a rug, therefore, it is named a rug plot.

[34]:
```
sns.rugplot(data=tips, x='total_bill', y='tip');
```



Similarly, we can combine `relplot()` and `rugplot()` in one figure.

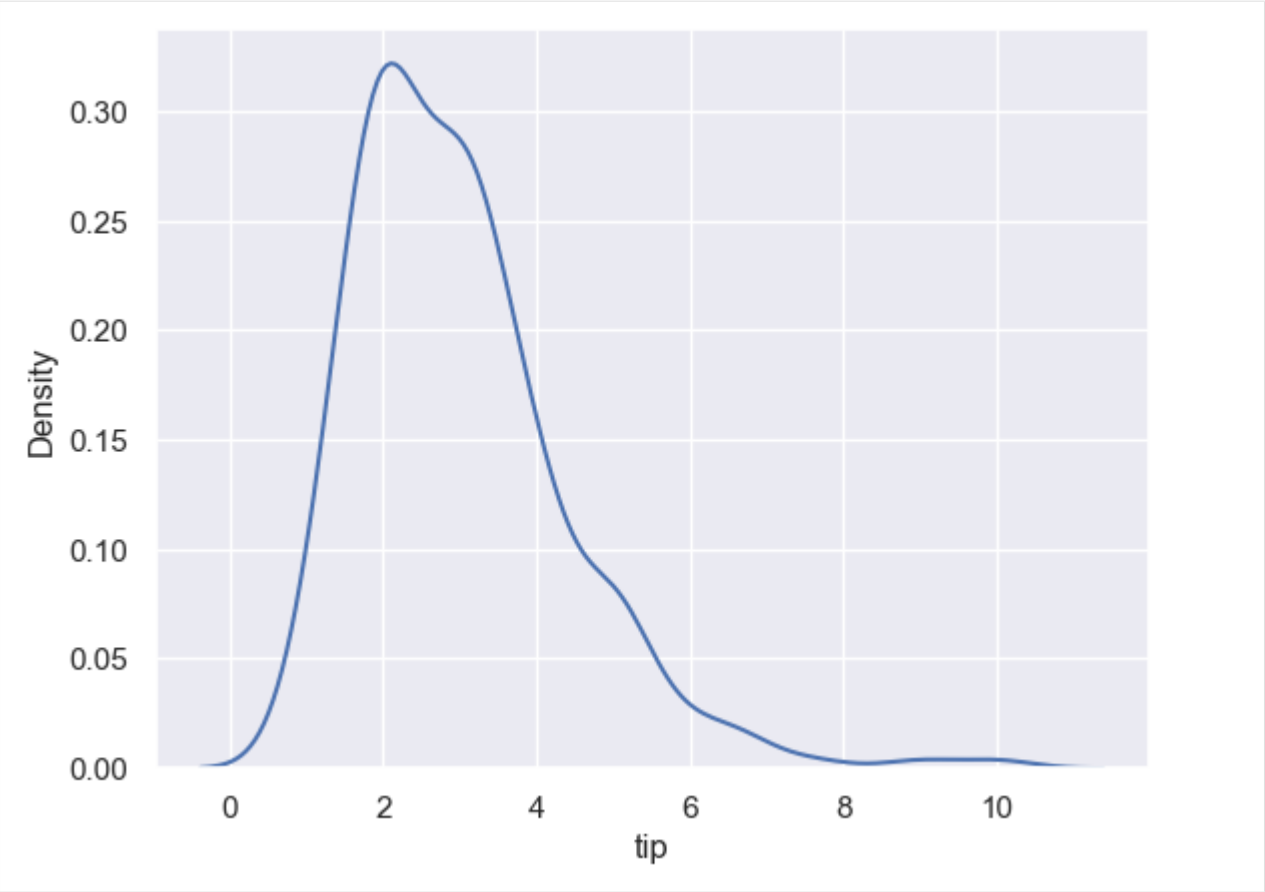
[35]:
```
sns.relplot(data=tips, x='total_bill', y='tip')
sns.rugplot(data=tips, x='total_bill', y='tip');
```



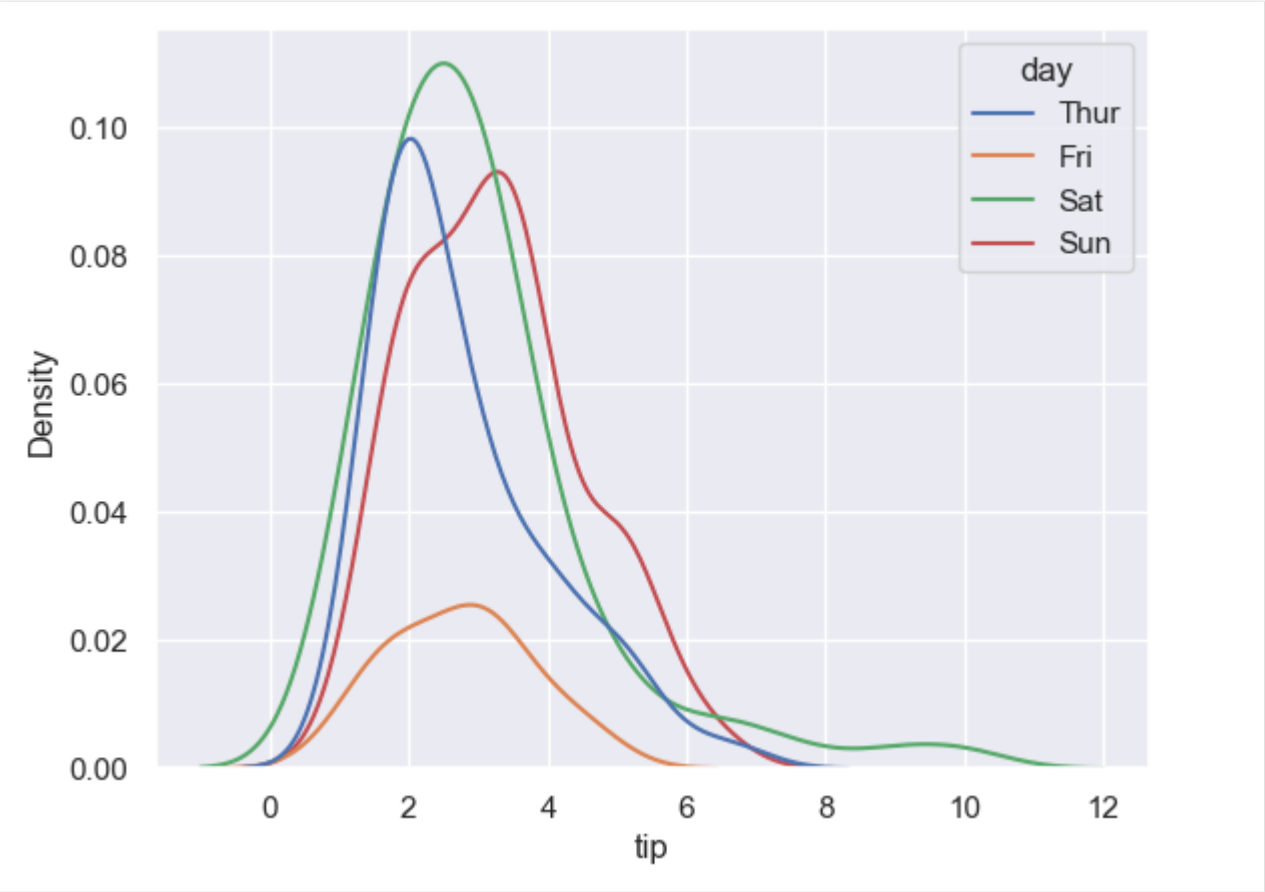
Kernel Density Estimation (KDE) Plot with `kdeplot()`

Besides using the `kde` parameter with `displot()`, Seaborn also provides the function `kdeplot()` to visualize the smoothed curve of the probability density function of a single feature or multiple features.

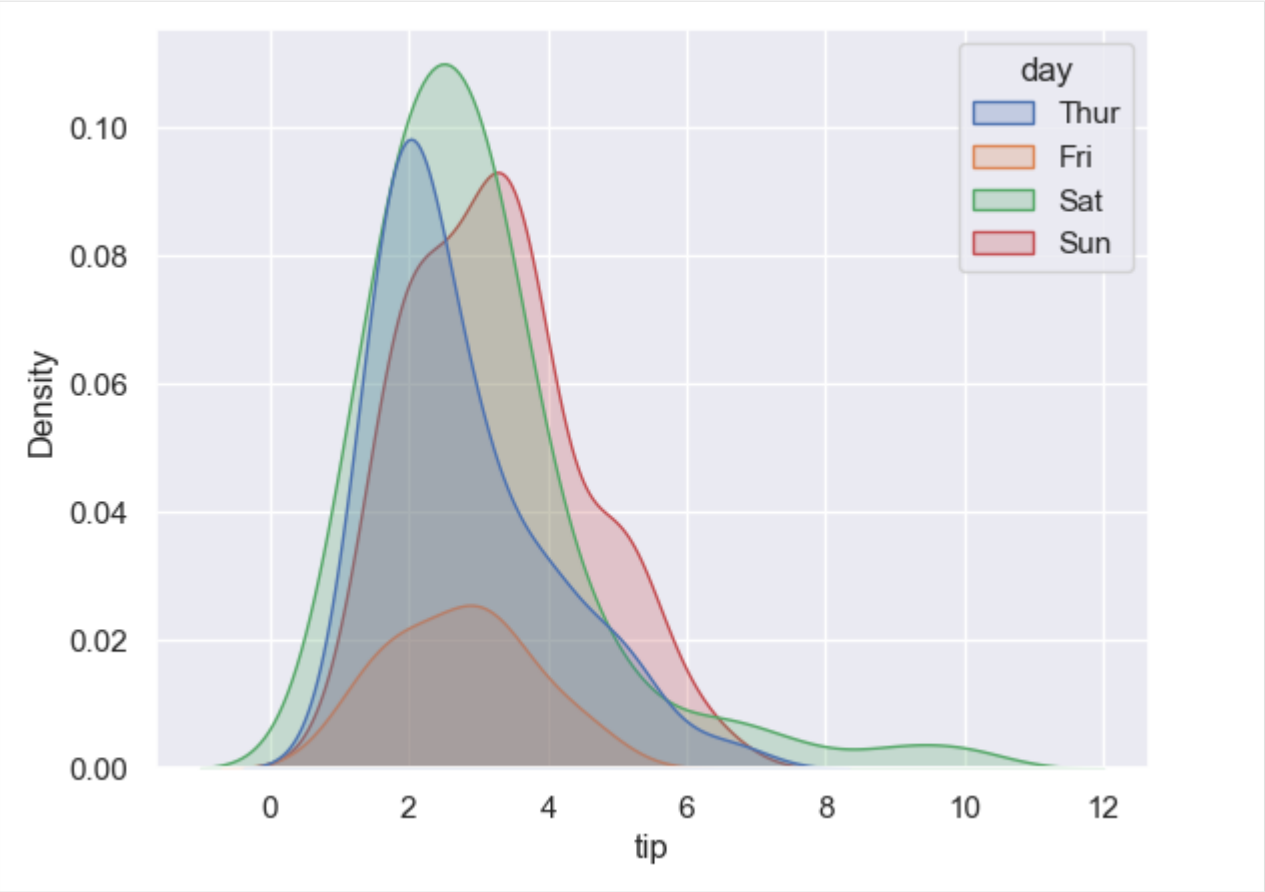
[36]:
```
sns.kdeplot(data=tips, x='tip');
```



[37]:
```
sns.kdeplot(data=tips, x='tip', hue='day');
```



[38]:
```
sns.kdeplot(data=tips, x='tip', hue='day', fill=True);
```

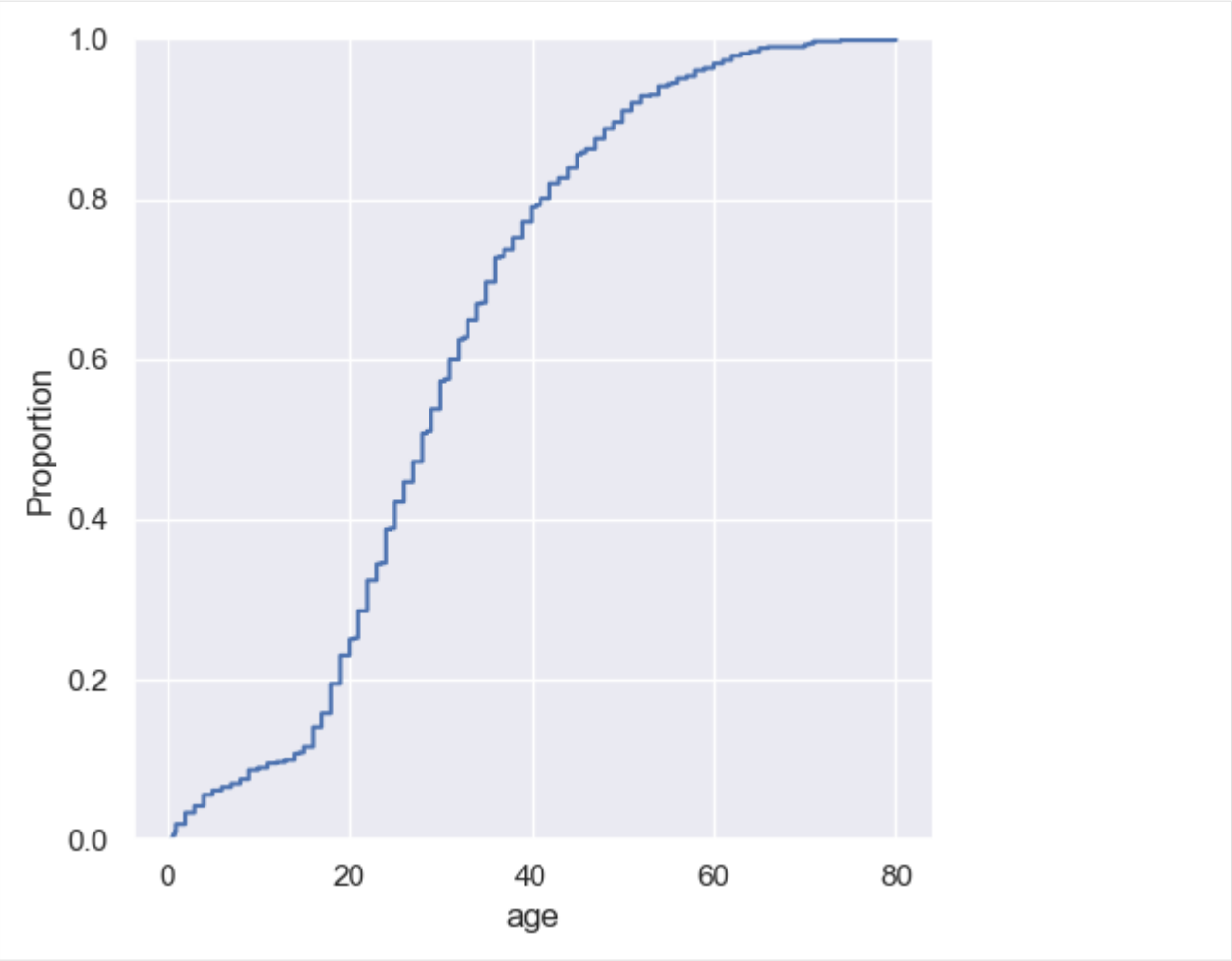


[39]:
```
sns.kdeplot(data=tips, x='tip', hue='day', multiple='stack');
```

Cumulative Distributions

By setting the parameter `kind` in `displot()` to `ecdf`, we can plot the cumulative distribution of a feature, where `ecdf` stands for `empirical cumulative distribution function`.

[40]:
```
sns.displot(titanic, x='age', kind='ecdf');
```



7.12.3 12.3 Categorical Plots

In Seaborn, there are various plot functions for visualizing categorical data. These include:

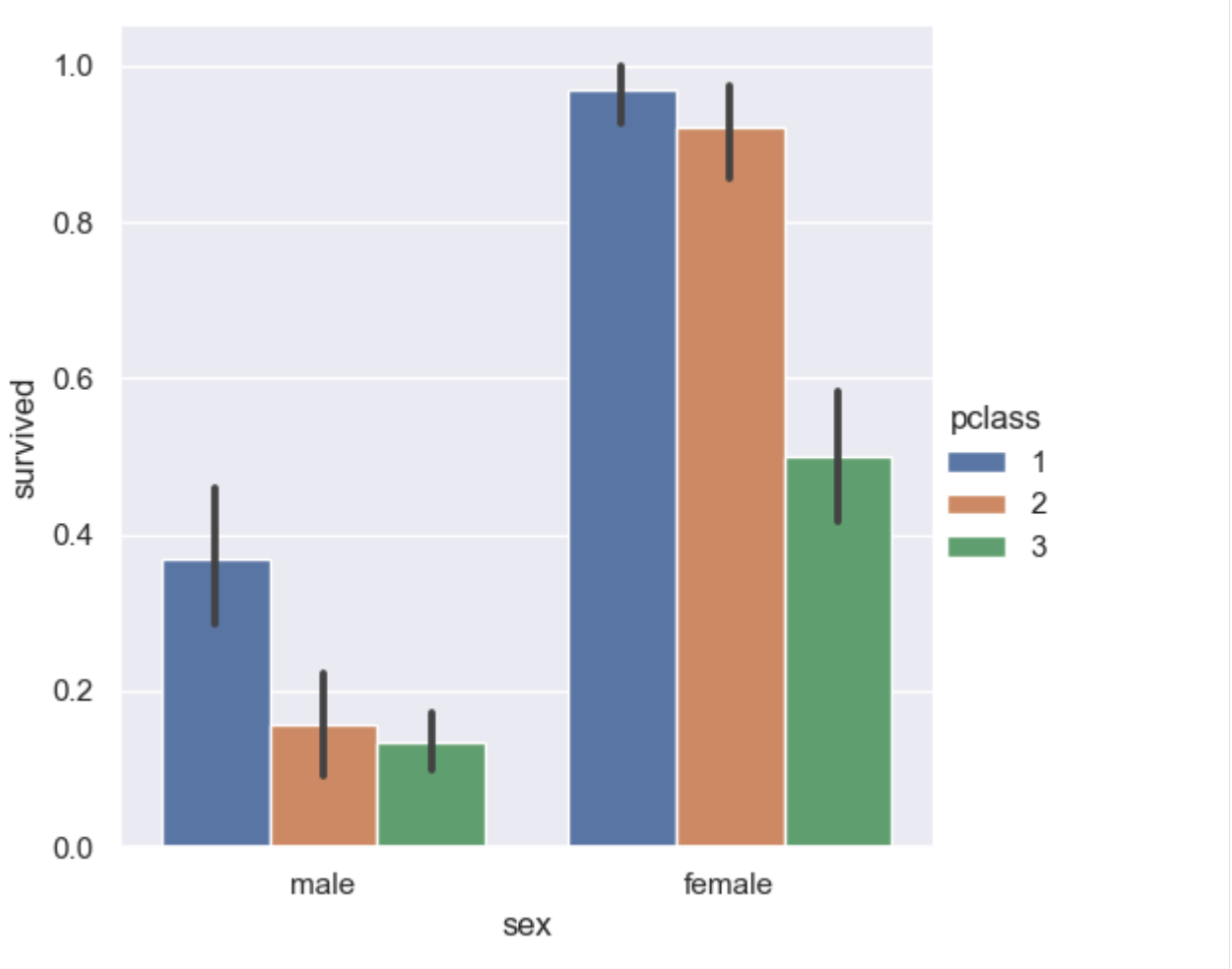
1. Categorical estimate plots
   - `barplot()`
   - `countplot()`
   - `pointplot()`
2. Categorical distribution plots
   - `boxplot()`
   - `boxenplot()`
   - `violinplot()`
3. Categorical scatter plots
   - `stripplot()`
   - `swarmplot()`

Similar to the high-level function `displot()` for creating distribution plots, Seaborn also provides a high-level function `catplot()` for plotting all of the above types of categorical plots, by passing different values for the `kind` parameter. The available settings include: `bar`, `count`, `point`, `box`, `boxen`, `violin`, `strip`, and `swarm`.

Categorical Estimate Plots

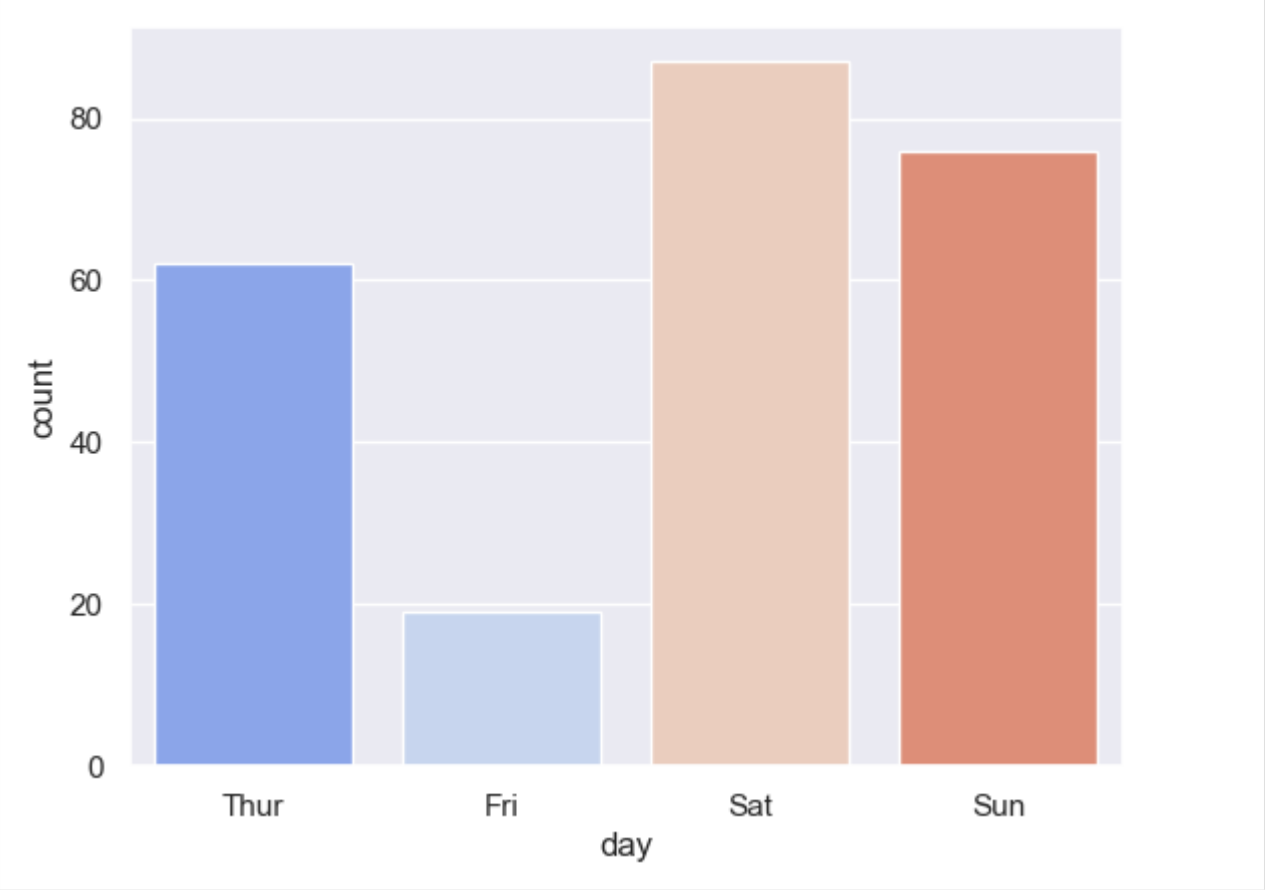
The function `barplot()` is used to visualize statistics of categorical data based on different statistical estimator functions (mean is the default estimate). As stated earlier, to create bar plots we can either use `barplot()` or `catplot(..., kind='bar')`.

[41]:
```
sns.catplot(data=titanic, x='sex', y='survived', hue='pclass', kind='bar');
```



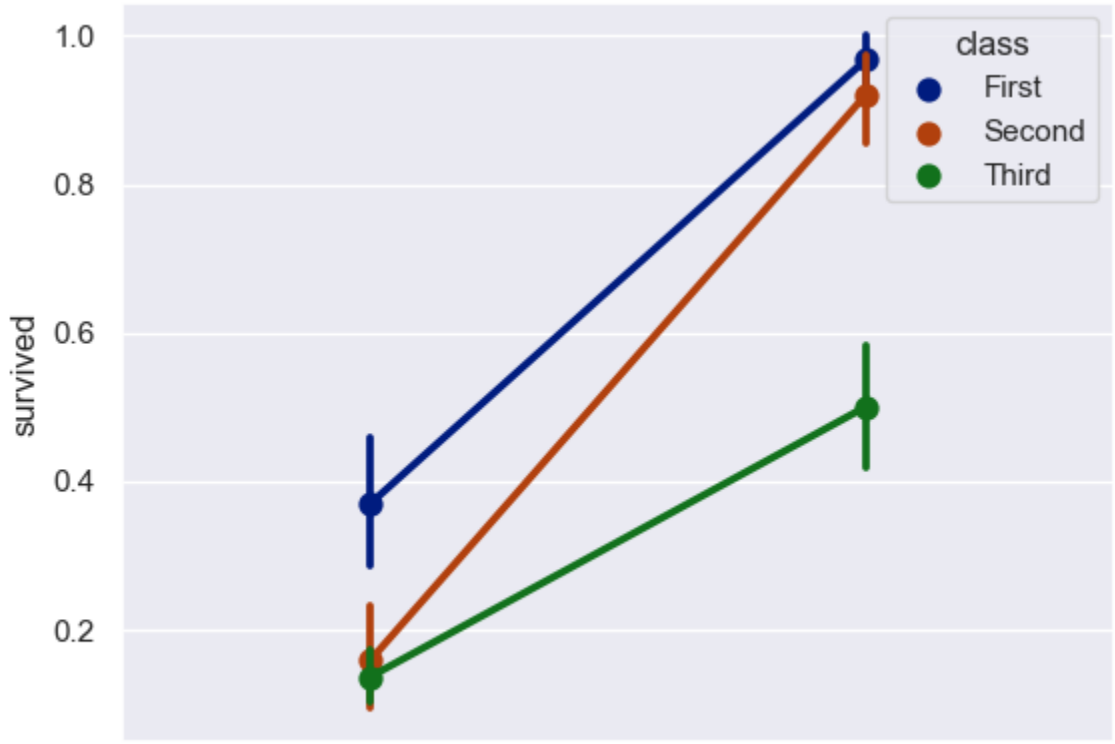
The function `countplot()` is used to visualize the number of observations in each category.

[42]:
```
sns.countplot(data=tips, x='day', palette='coolwarm');

# The same plot can be obtained with
#sns.catplot(data=tips, x='day', palette='coolwarm', kind='count')
```



Rather than plotting bars, the function `pointplot()` visualizes the point estimates of categorical data. Notice that it can also connect the points with the categorical variable specified with `hue`.

[43]:
```
sns.pointplot(data=titanic, x='sex', y='survived', hue='class', palette='dark');
```

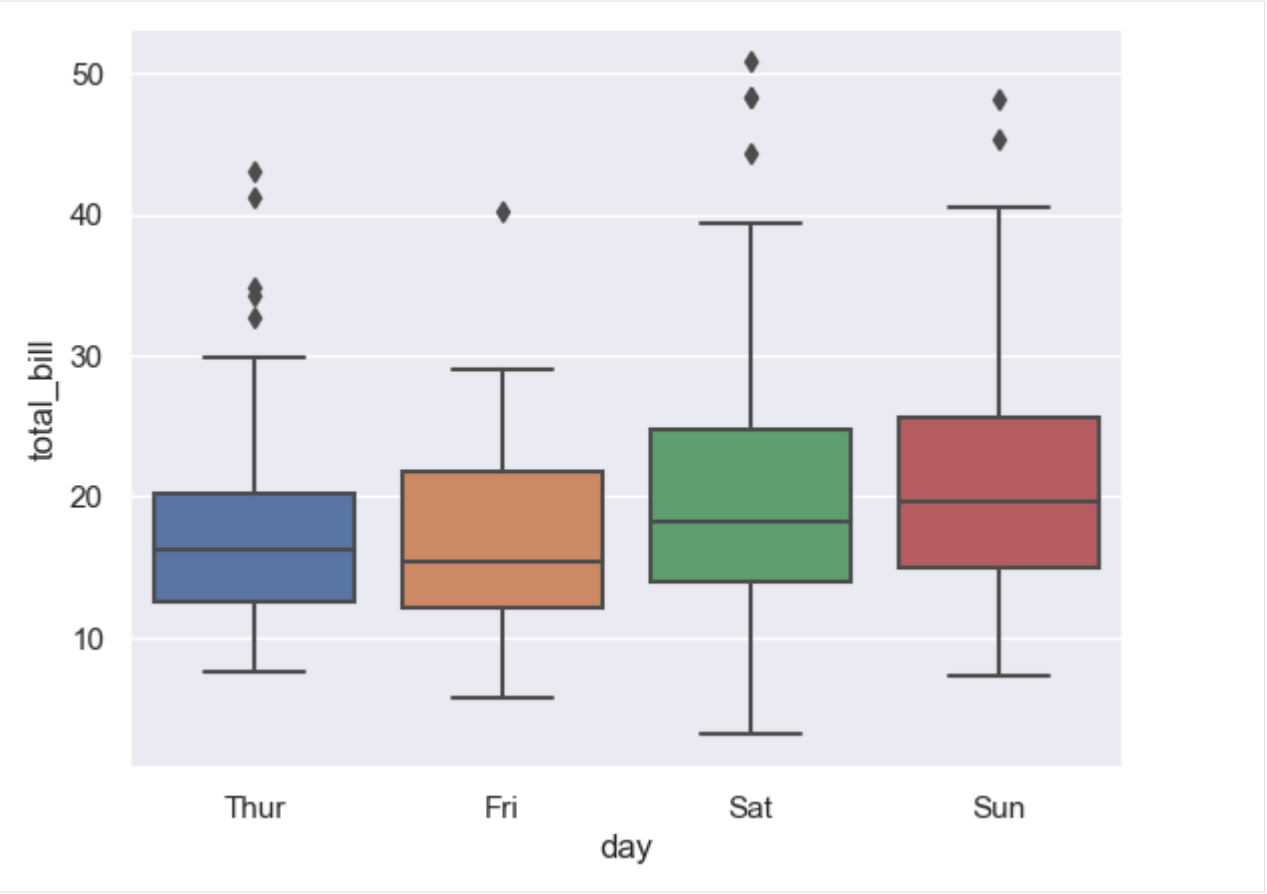


Categorical Distribution Plots

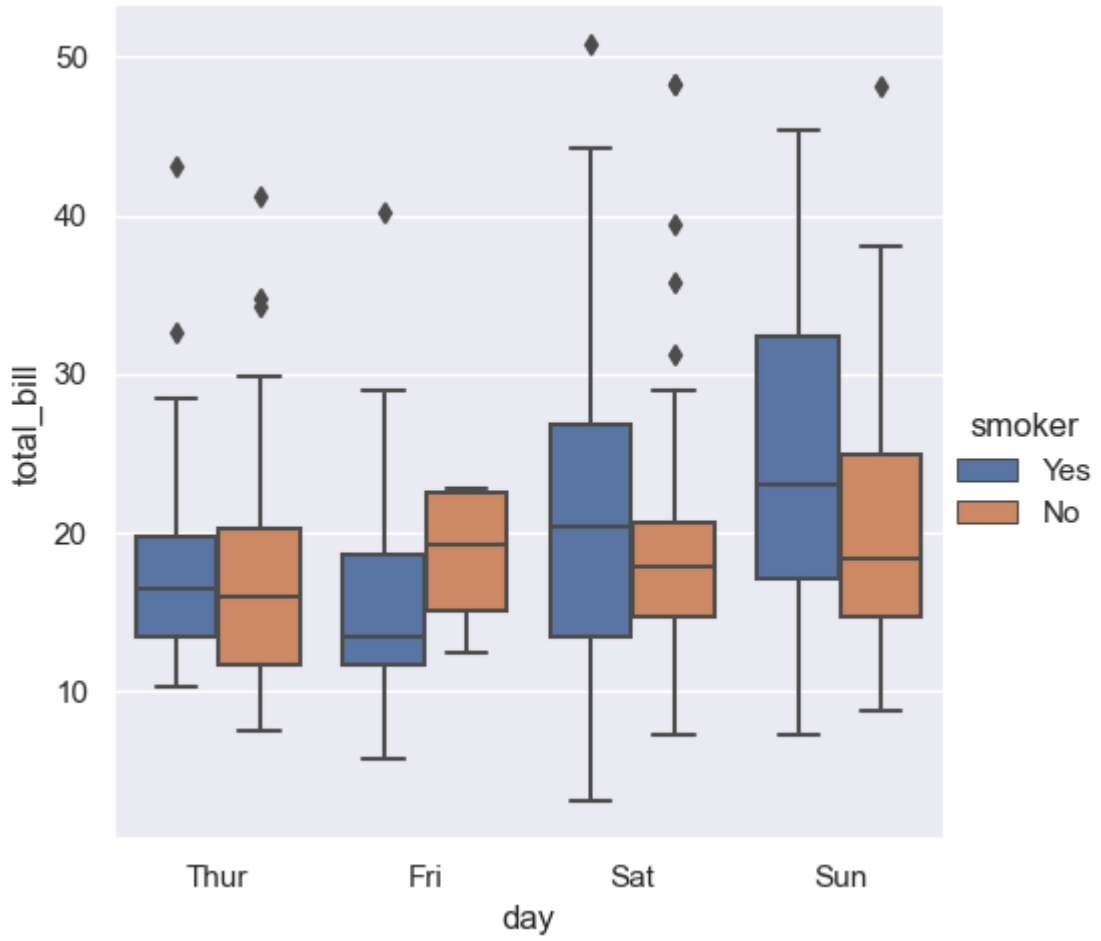
The functions `boxplot()`, `boxenplot()`, and `violinplot()` are used to plot the distributions of categorical data.

A box plot (or box-and-whisker plot) shows the median, the inter-quartile range with the first (25%) and third (75%) quartiles of the dataset, and the whiskers extend to show the rest of the distribution. Exceptions are data points that are determined to be outliers based on a method that is a function of the inter-quartile range.

[44]:
```
sns.boxplot(data=tips, x='day', y='total_bill');
```



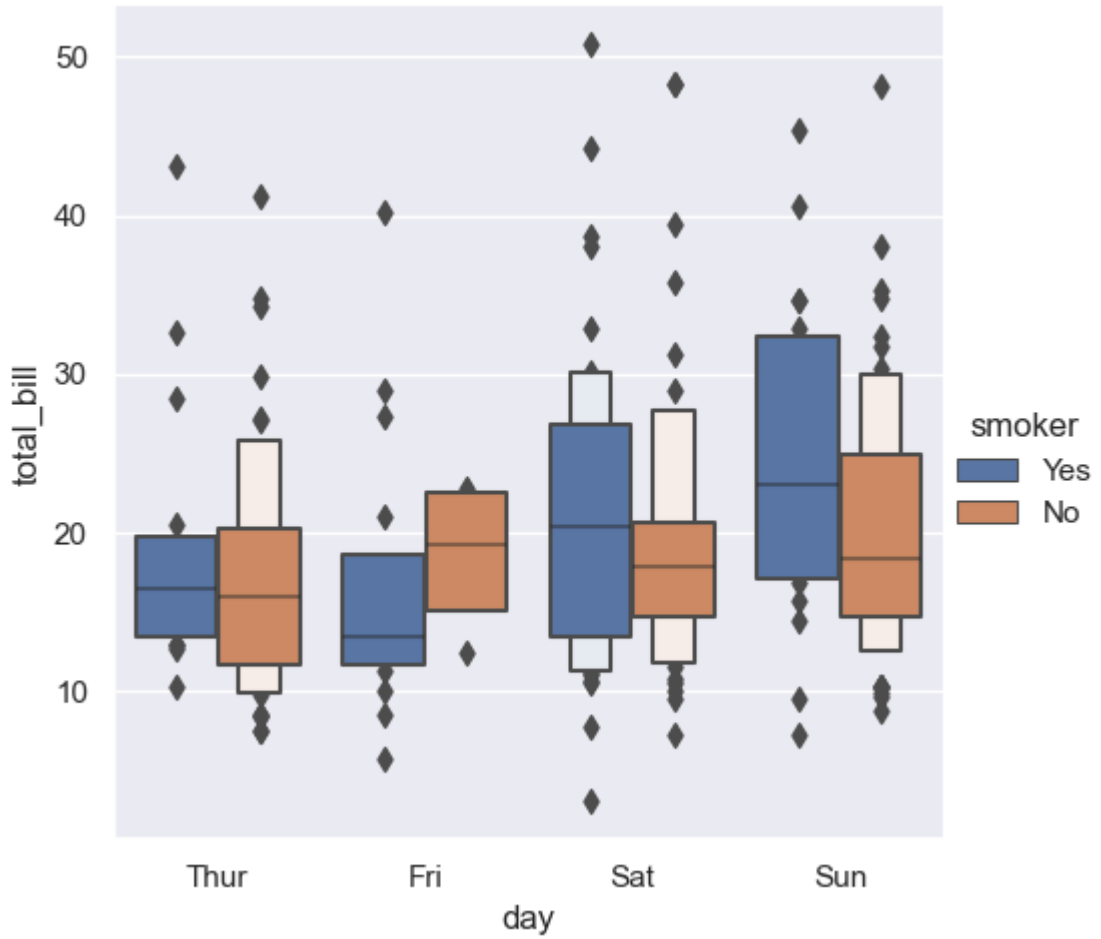
[45]:
```
sns.catplot(data=tips, x='day', y='total_bill', hue='smoker', kind='box');
```



The `kind` parameter set to `boxenplot` creates boxen plots, which are an enhanced version of box plots. Boxen plots employ additional steps for calculating the quartiles, and are more efficient in visualizing features that have many outliers.

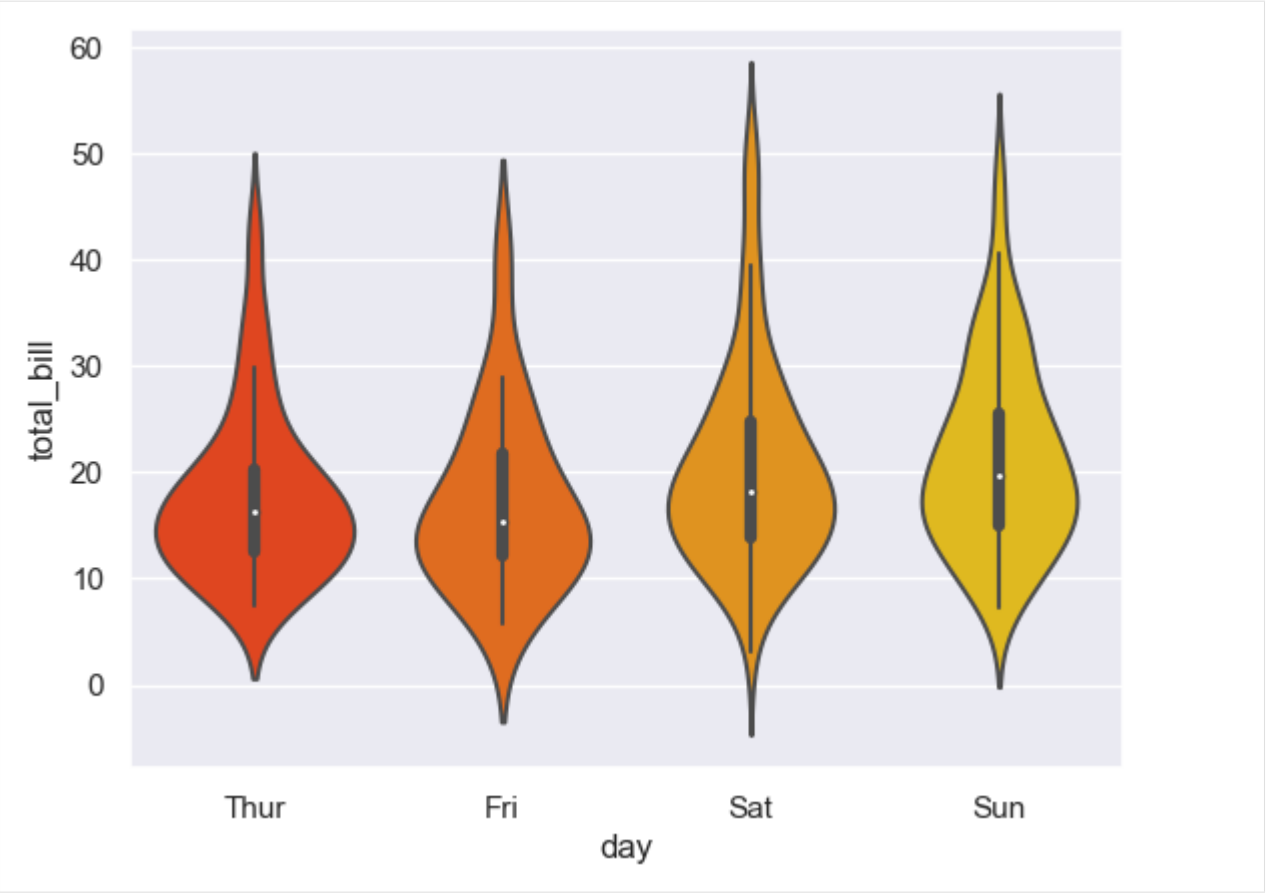
The approach for creating boxen plots includes first dividing the data into groups and then iteratively constructing the plot by starting with the largest group and data and progressing to the smallest group of data. The boxes can be considered subgroups of data, with the width and height corresponding to the size and range of the data in the groups. Therefore, the boxes do not correspond to the first and third quartiles, as in traditional box plots.

[46]:
```
sns.catplot(data=tips, x='day', y='total_bill', hue='smoker', kind='boxen');
```



The function `violinplot()` shows the entire distribution of categorical data and uses a rotated kernel density plot on both the right and left sides of the plots. Understandably, the plots resemble violins, and they make it easier to visualize the shape, skewness, and presence of multiple peaks in the data distribution. Unlike box plots and boxen plots, violin plots do not display the individual outliers in the data.

[47]:
```
sns.violinplot(data=tips, x='day', y='total_bill', palette='autumn');
```

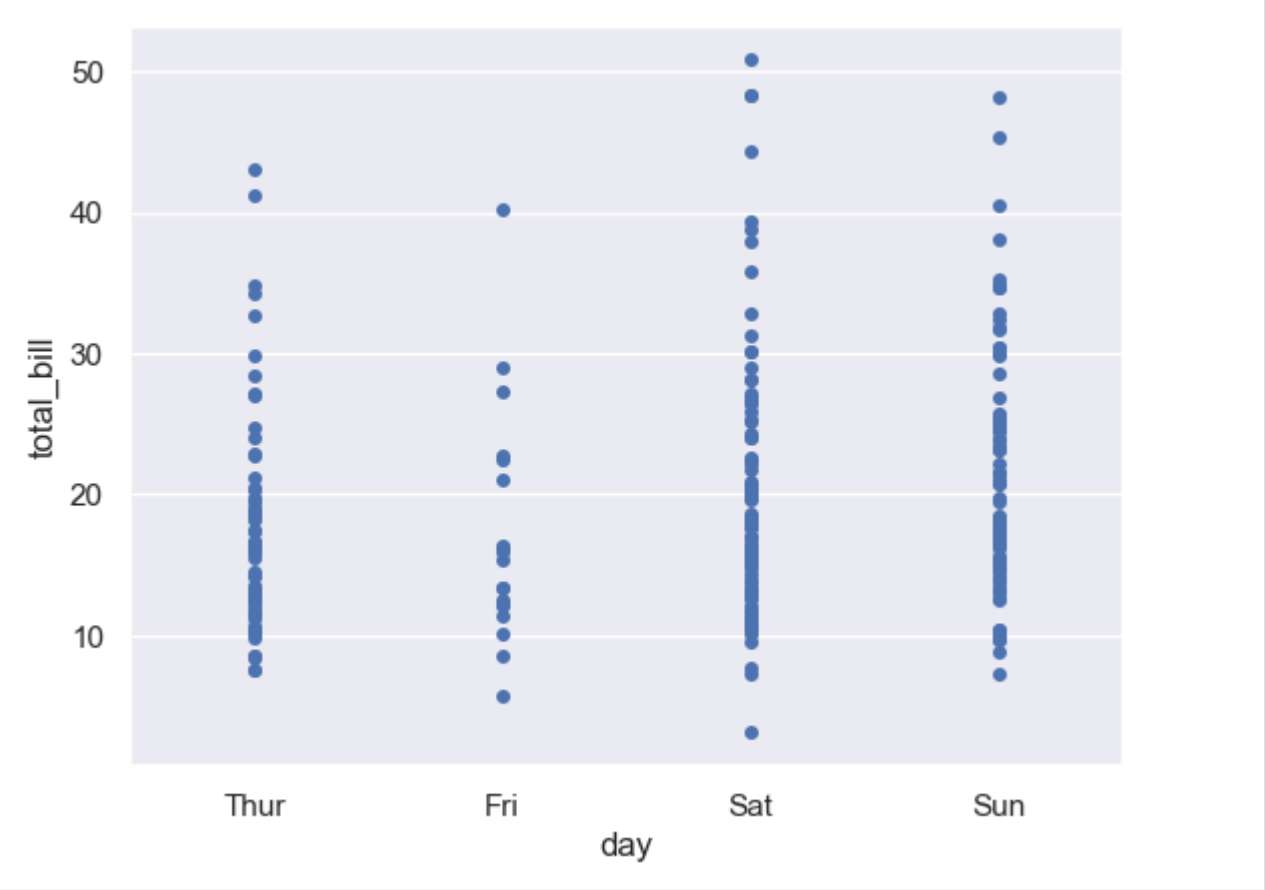


Categorical Scatter Plots

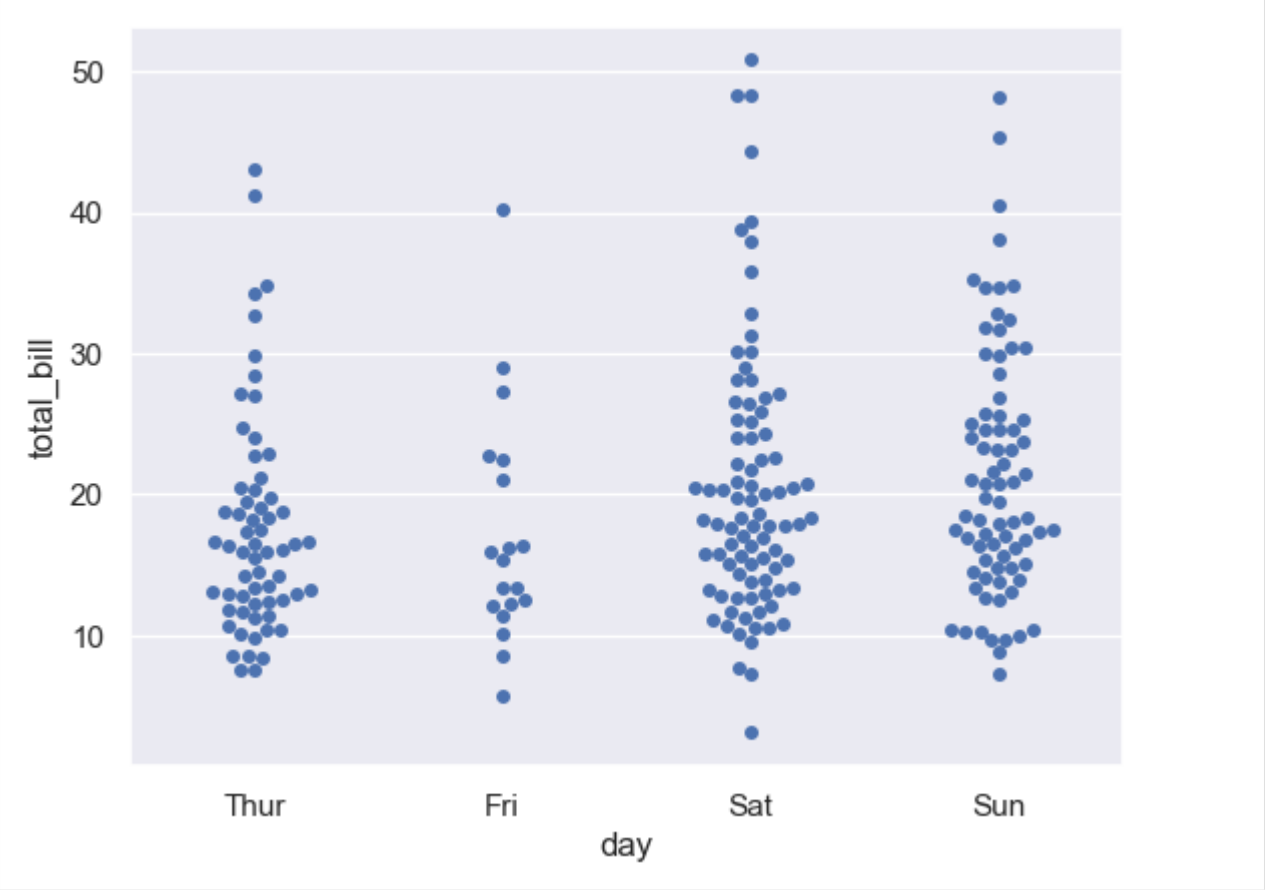
The functions `stripplot()` and `swarmplot()` visualize categorical data with scatter plots.

In a `stripplot()` the data points are shown in a strip, whereas in a `swarmplot()` the data are randomly shifted horizontally to avoid overlaps. Strip plots are simpler and can be used to see the exact positions of data points, and swarm plots resemble a swarm of bees and they make it easier to see the distribution of the data points.

[48]:
```
sns.stripplot(data=tips, x='day', y='total_bill', jitter=False);
```

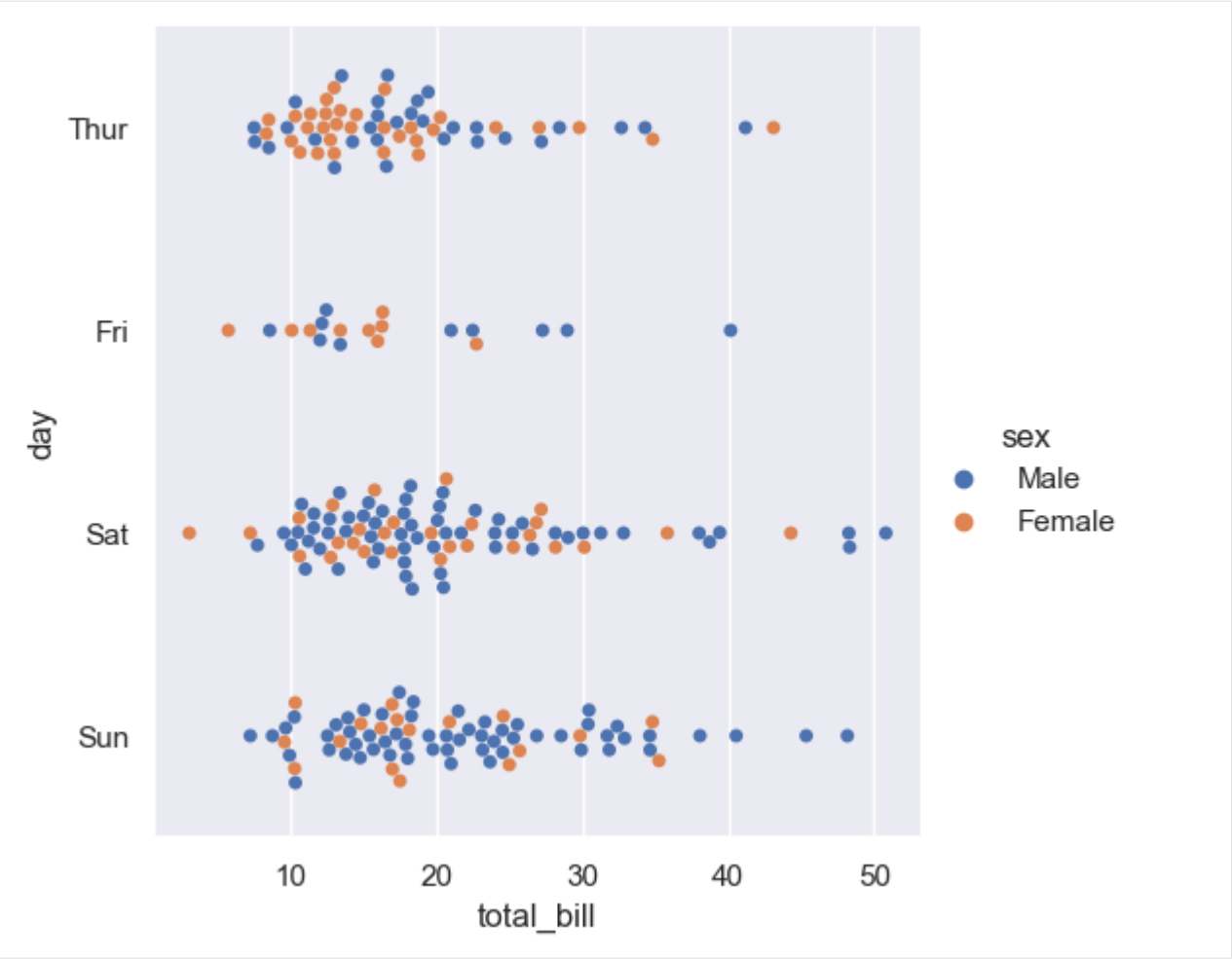


[49]:
```
sns.swarmplot(data=tips, x='day', y='total_bill');
```



Also, when using `catplot()`, the parameter `strip` is the default value of the `kind` parameter.

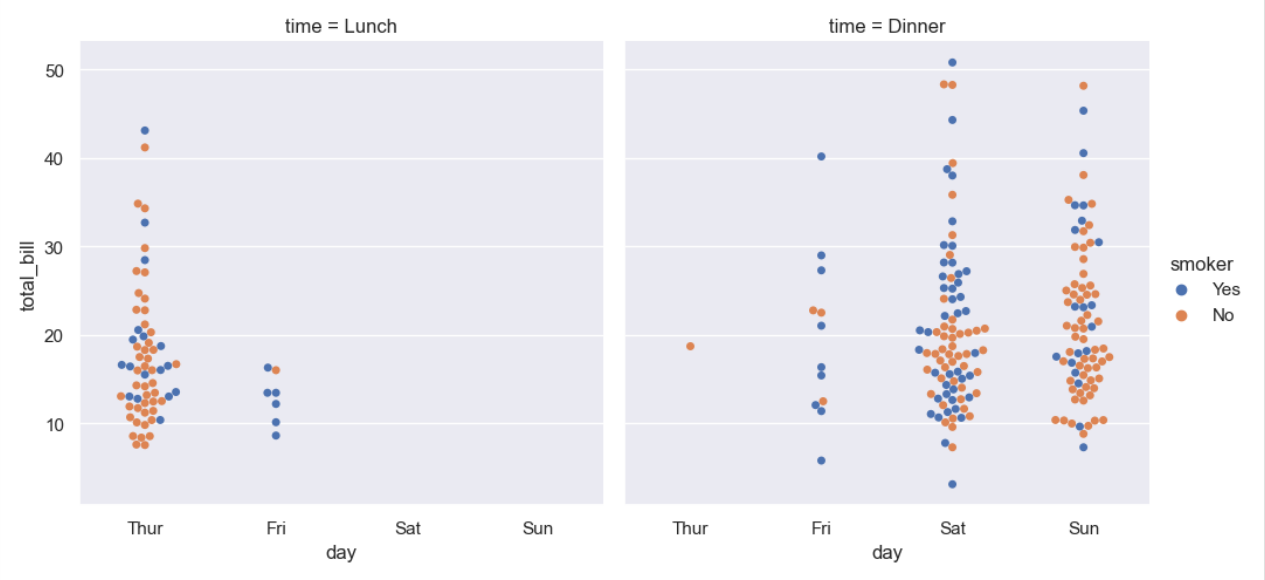
[50]:
```
sns.catplot(data=tips, x='total_bill', y='day', hue='sex', kind='swarm');
```



Plotting Multiple Categorical Plots

We can plot multiple plots with `catplot()` by setting the parameters `col` or `row` to a data feature.

[51]:
```
sns.catplot(data=tips, x='day', y='total_bill', hue='smoker', col='time', kind='swarm');
```

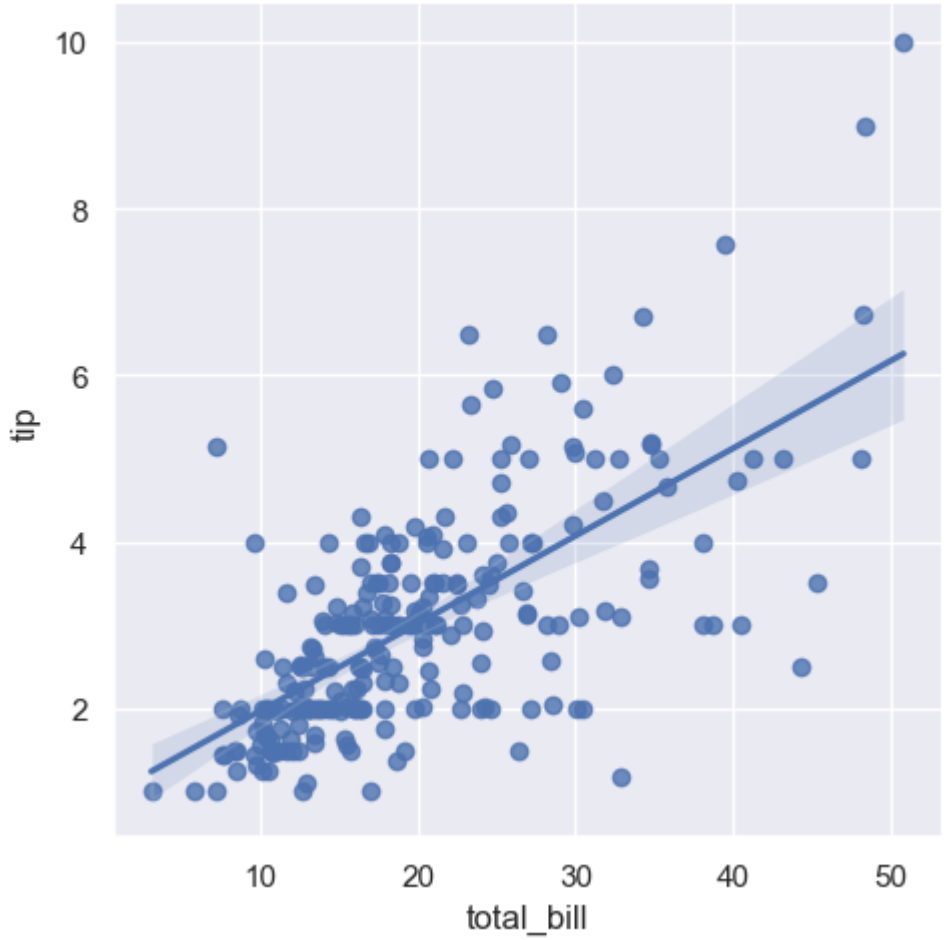


7.12.4 12.4 Regression Plots

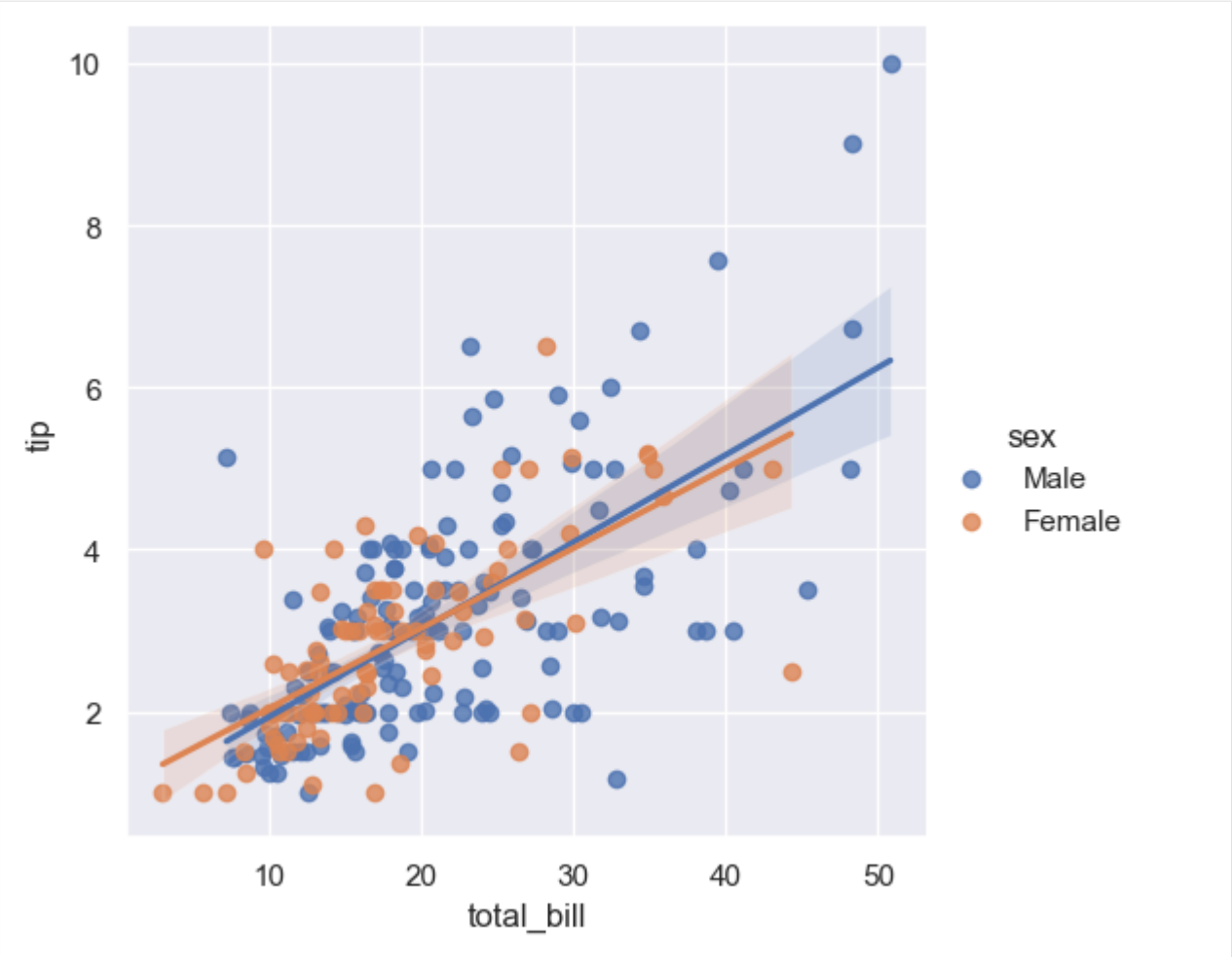
Seaborn provides plots for visualizing the linear relationship between the features. The functions `regplot()` and `lmplot()` can be used for fitting a linear regression model, where `lm` stands for linear model.

For instance, the figure below shows the regression line, as well as the uncertainty in the predicted values corresponding to 95% confidence interval for the regression fit with `lmplot()`.

[52]:
```
sns.lmplot(data=tips, x='total_bill', y='tip');
```



[53]:
```
sns.lmplot(data=tips, x='total_bill', y='tip', hue='sex');
```

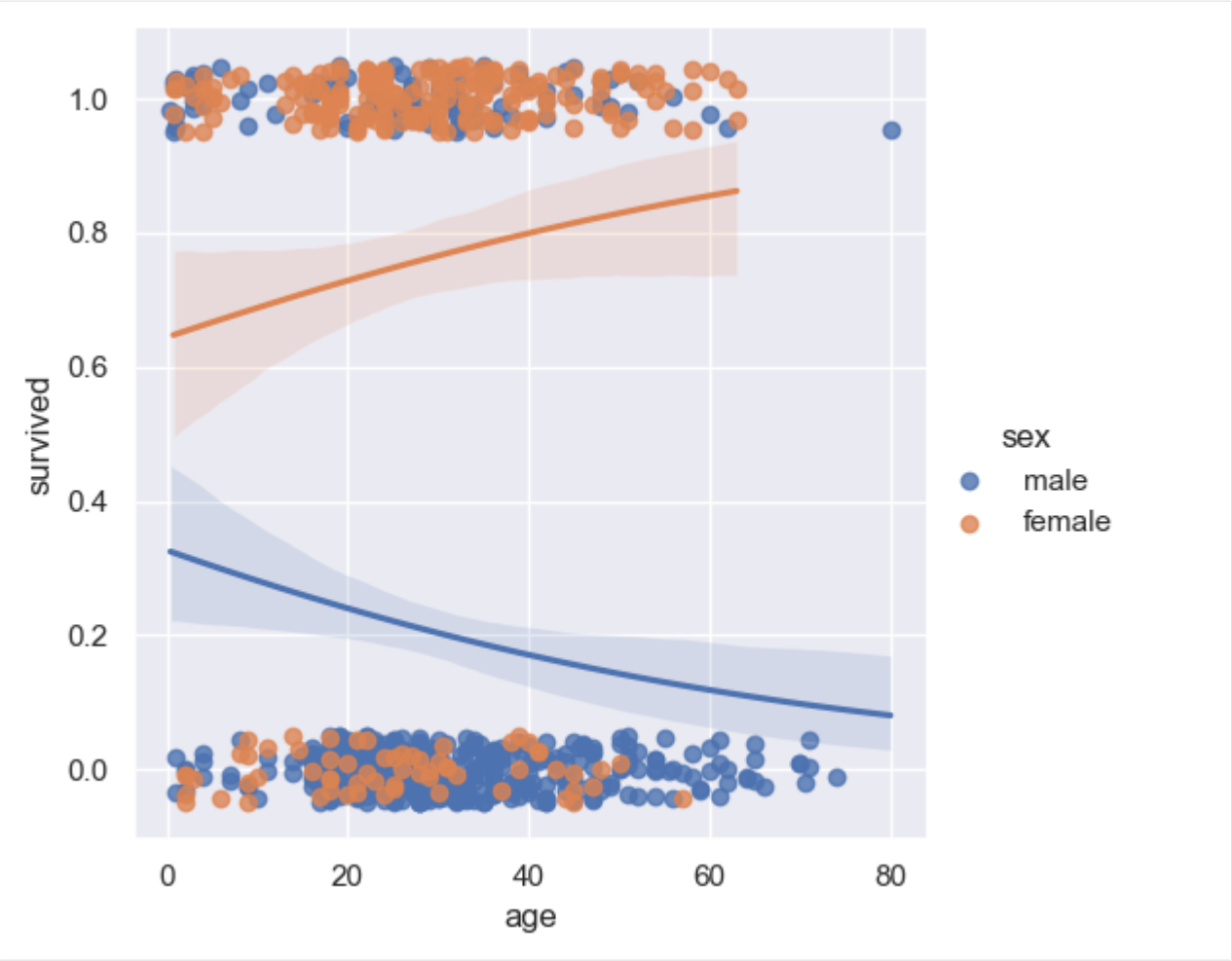


If one of the features is a categorical variable (e.g., `survived` in the next example), linear regression models may not produce good results, and an alternative is to apply a logistic regression model by using `logistic=True`. The regression line shown the estimated probability of the variable `y=1` for a given value of `x`.

The function `lmplot()` also allows to add jitter to the data in the scatter plot by using the `y_jitter` parameter, in order to better visualize the values. Jitter is applied only to the scatter plot, and it is not applied to the data for fitting the regression model.

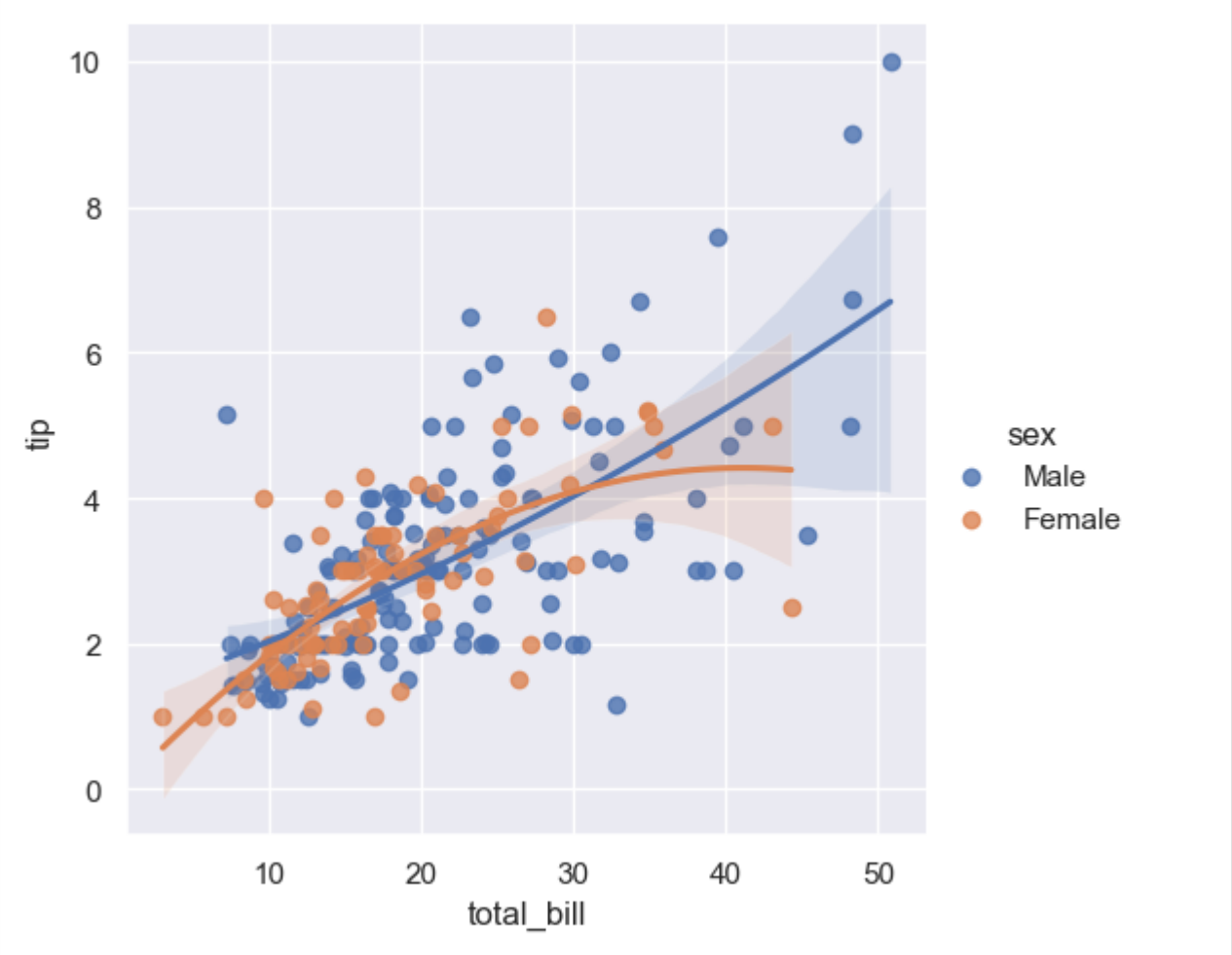
In the plot below, the regression lines indicate that the probablity of men to survive decreased with age, whereas the probability of women to survive increased with age.

[54]:
```
sns.lmplot(data=titanic, x='age', y='survived', logistic=True, y_jitter=0.05, hue='sex');
```



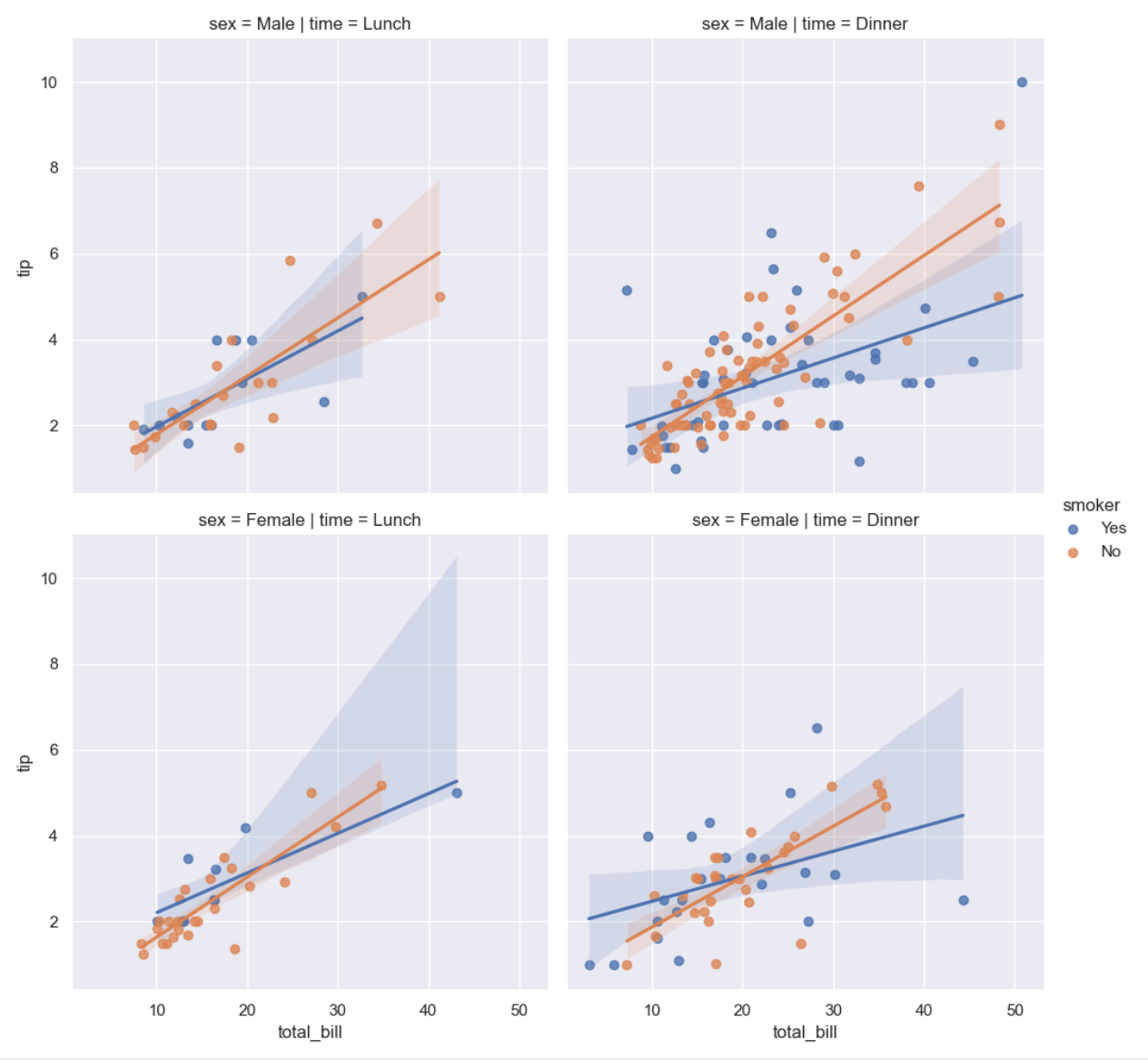
If the relationship between two features is not linear, the function `lmplot()` can also fit a polynomial regression model to identify nonlinear trends in the dataset by using the parameter `order`. E.g., `order=2` in the next cell fits a second-degree polynomial regression model.

[55]:
```
sns.lmplot(data=tips, x='total_bill', y='tip', hue='sex', order=2);
```



As we explained in the above sections, we can create subplots by setting the parameters `col` and `row` to specific data features.

[56]:
```
sns.lmplot(data=tips, x='total_bill', y='tip', hue='smoker', col='time', row='sex');
```



7.12.5 12.5 Multiple Plots

Multiple plot functions in Seaborn are used to visualize multiple features along different axes. Seaborn provides the following functions:

- `FacetGrid()`
- `PairGrid()`
- `PairPlot()`

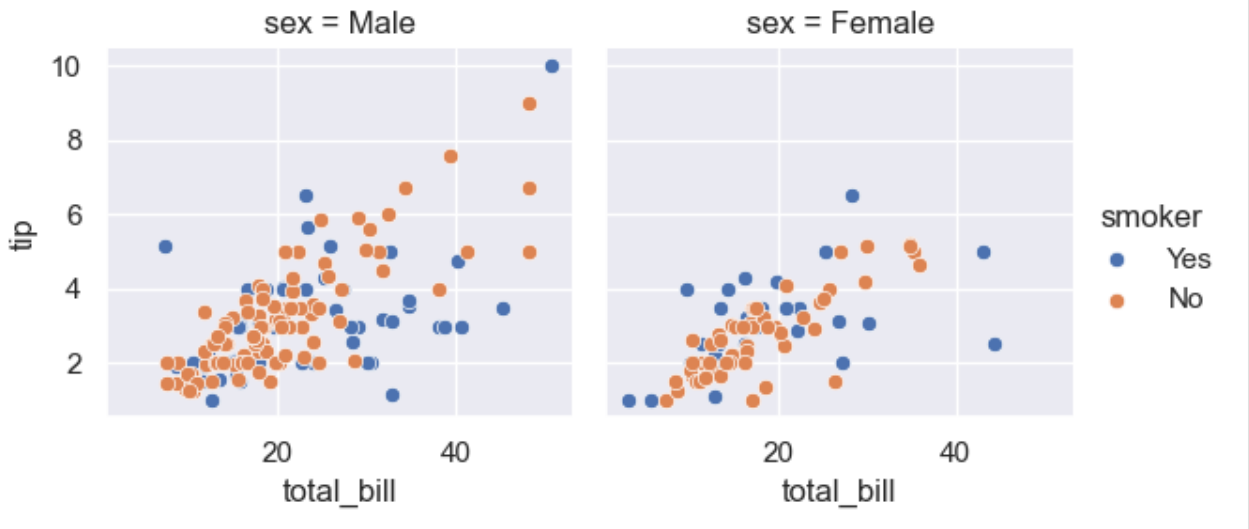
FacetGrid()

The function FacetGrid() creates multiple grid plots and allows plotting the features on row and column axes in order to visualize the relationship between multiple features in separate subplots. It allows to draw a grid based on three parameters: col, row, and hue, where hue allows to draw different categories in a subplot with separate colors.

After the grid is created with FacetGrid(), visualizations are added with the map() method that specifies the plot type (such as scatter, histogram, bar plot, etc.).

[57]:
```
plot = sns.FacetGrid(tips, col='sex', hue='smoker')
```

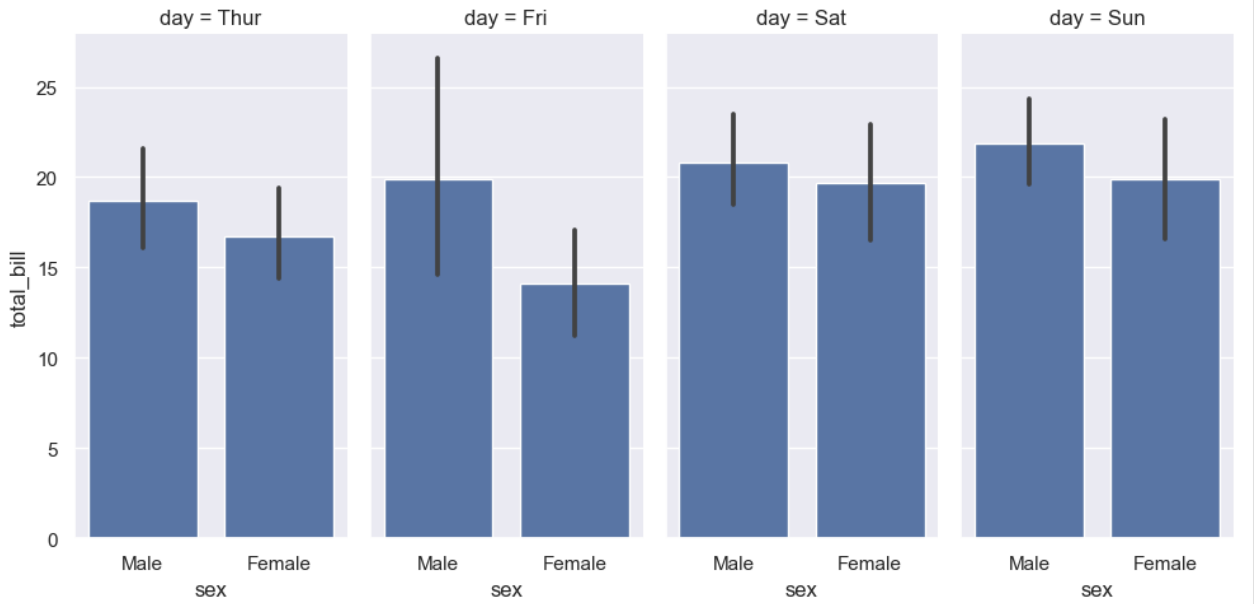
[58]:
```
plot = sns.FacetGrid(tips, col='sex', hue='smoker')
plot.map(sns.scatterplot,'total_bill', 'tip')
plot.add_legend();
```



FacetGrid() allows to control the height, aspect ratio, and other properties of the subplots in the grid.

[59]:
```
plot = sns.FacetGrid(tips, col='day', height=5, aspect=0.5)
plot.map(sns.barplot, 'sex', 'total_bill');
```

```
C:\Users\vakanski\anaconda3\Lib\site-packages\seaborn\axisgrid.py:712: UserWarning:␣
↪Using the barplot function without specifying `order` is likely to produce an␣
↪incorrect plot.
  warnings.warn(warning)
```



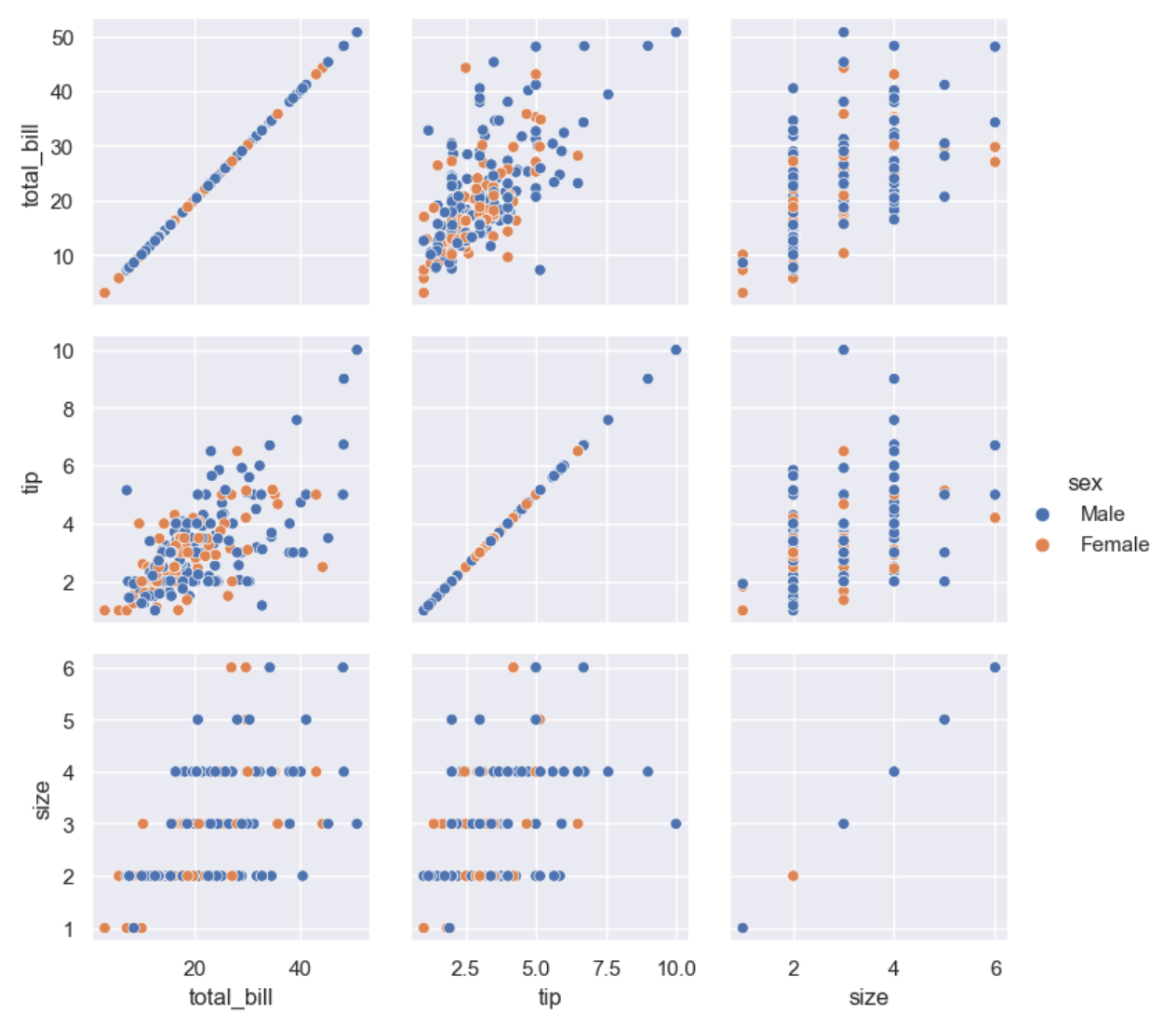
PairGrid()

The function `PairGrid()` allows to plot pairwise relationships between the features in the dataset. It is similar to the function `pairplot()` that we saw earlier for multivariate data analysis.

An example is shown next, where `PairGrid` creates a grid of subplots by assigning each row and column to a different feature in the dataset. Since the `tips` dataset has three numerical columns, the created grid is of 3x3 size.

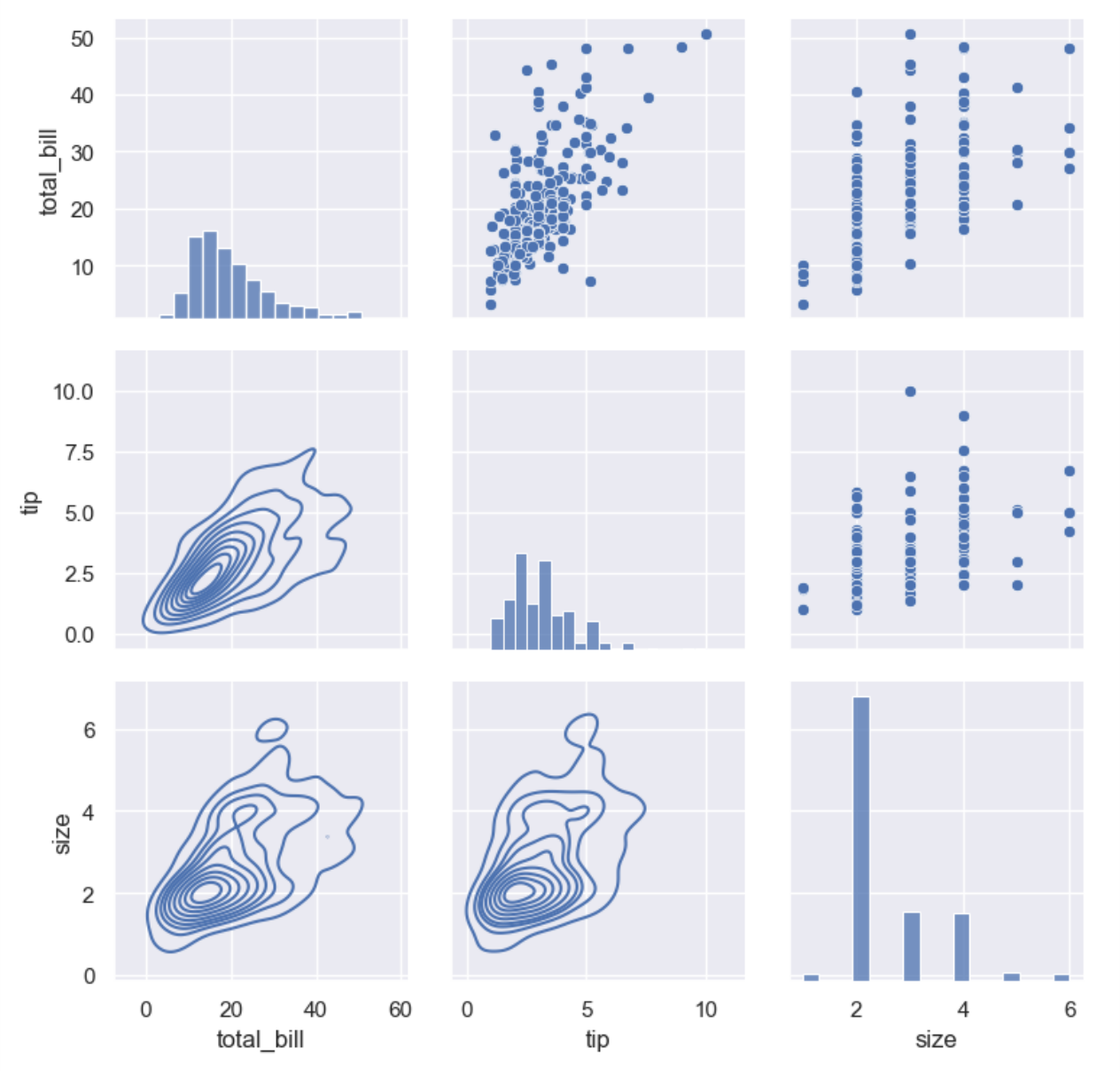
Note also the difference between `FacetGrid()` and `PairGrid()`. `FacetGrid()`shows the same relationship between two features, conditioned on different categories of other features. E.g., in the above example, all subplots show the scatter plots of `total_bill` and `tip`, conditioned on the features `sex` and `smoker`. `PariGrid()` shows different relationships between the features in the subplots.

[60]:
```
plot = sns.PairGrid(tips, hue='sex')
plot.map(sns.scatterplot)
plot.add_legend();
```



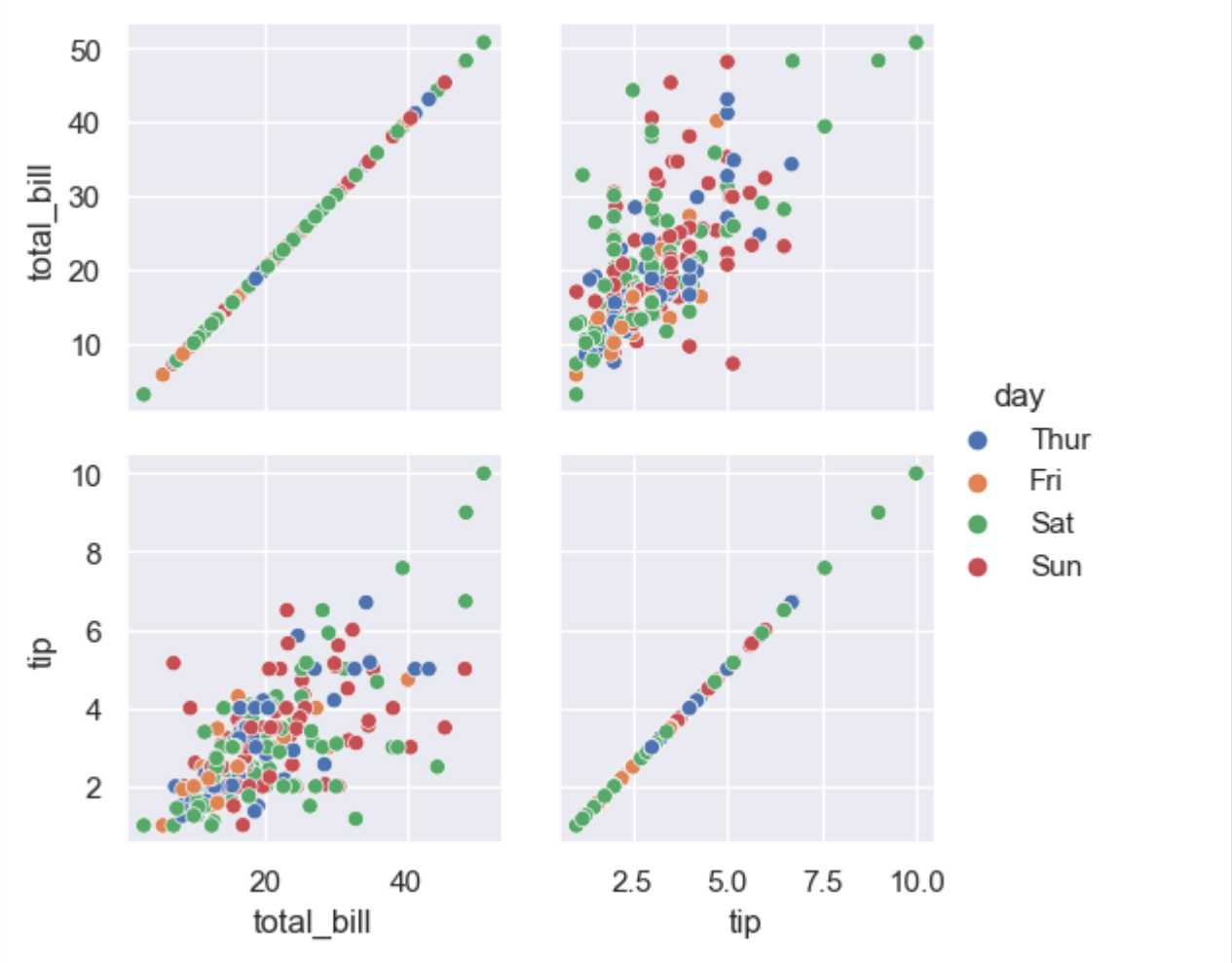
With `PairGrid()`, we can also be selective on the type of plots to include in diagonal subplots with `map_diag` and in the off-diagonal subplots (e.g., `map_upper` for the upper off-diagonal subplots and `map_lower` for the lower off-diagonal subplots).

[61]:
```
plot = sns.PairGrid(tips)
plot.map_diag(sns.histplot)
plot.map_upper(sns.scatterplot)
plot.map_lower(sns.kdeplot)
plot.add_legend();
```



Also, instead of plotting all features in a dataset, we can select the features we are interested in.

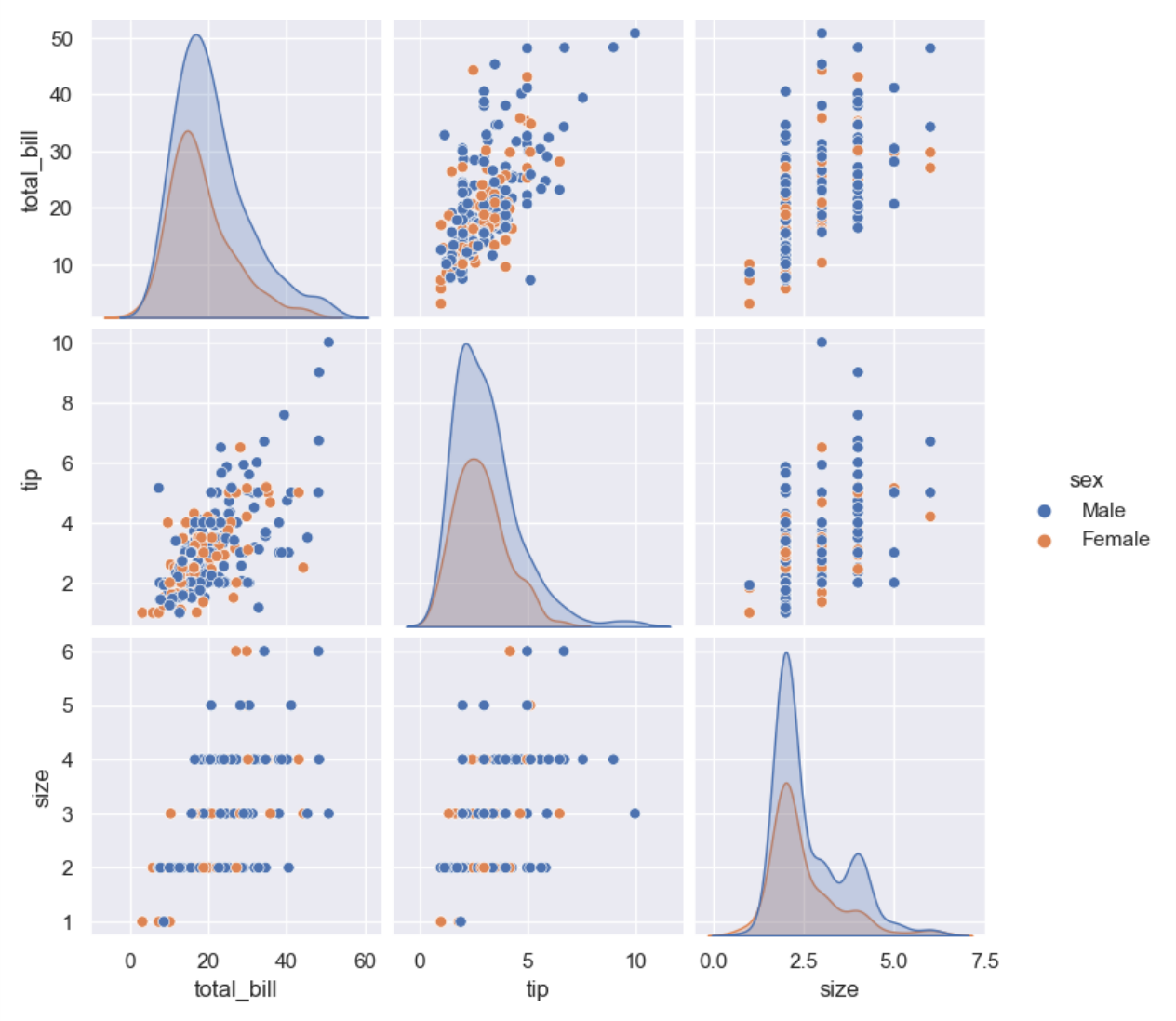
[62]:
```
plot = sns.PairGrid(tips, vars=['total_bill', 'tip'], hue='day')
plot.map(sns.scatterplot)
plot.add_legend();
```



pairplot()

The function `pairplot()` is very similar to `PairGrid()`, and it allows to plot pairwise relationships between the features in a dataset. It provides faster plotting in comparison to `PairGrid()` but it also offers less functionality.

[63]:
```
sns.pairplot(tips, hue='sex');
```



7.12.6 12.6 Matrix Plots

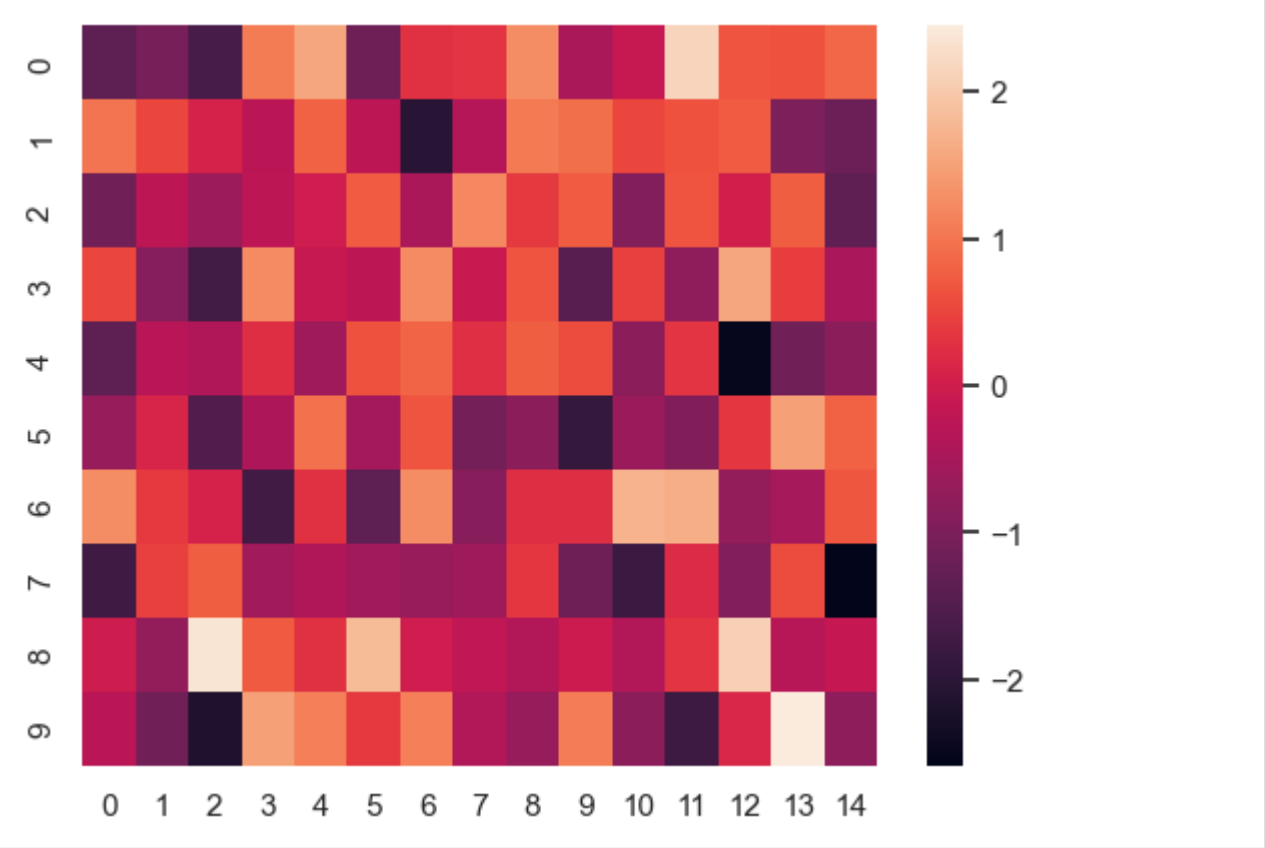
When performing data analysis, in some cases it is beneficial to visualize array data as color-encoded matrices that can be used to find patterns within the data.

Heat Maps

The function `heatmap()` in Seaborn adds colors to the elements in a matrix, and attaches a color bar to the plot.

Let's visualize a matrix with random values.

[64]:
```
sns.set_theme()
data = np.random.randn(10,15)
sns.heatmap(data);
```



Or, if you recall from the previous lecture, we used the `.corr()` method in pandas to calculate the correlation between the features, and afterward we used `heatmap` to visualize the correlations. By applying colors that correspond to the correlation values, it makes it easier to identify relationships between the features.

[65]:
```
# Finding the correlation of features in titanic dataset
correlation = titanic.corr()
correlation
```

```
C:\Users\vakanski\AppData\Local\Temp\ipykernel_8840\971058138.py:2: FutureWarning: The␣
↪default value of numeric_only in DataFrame.corr is deprecated. In a future version, it␣
↪will default to False. Select only valid columns or specify the value of numeric_only␣
↪to silence this warning.
  correlation = titanic.corr()
```

[65]:

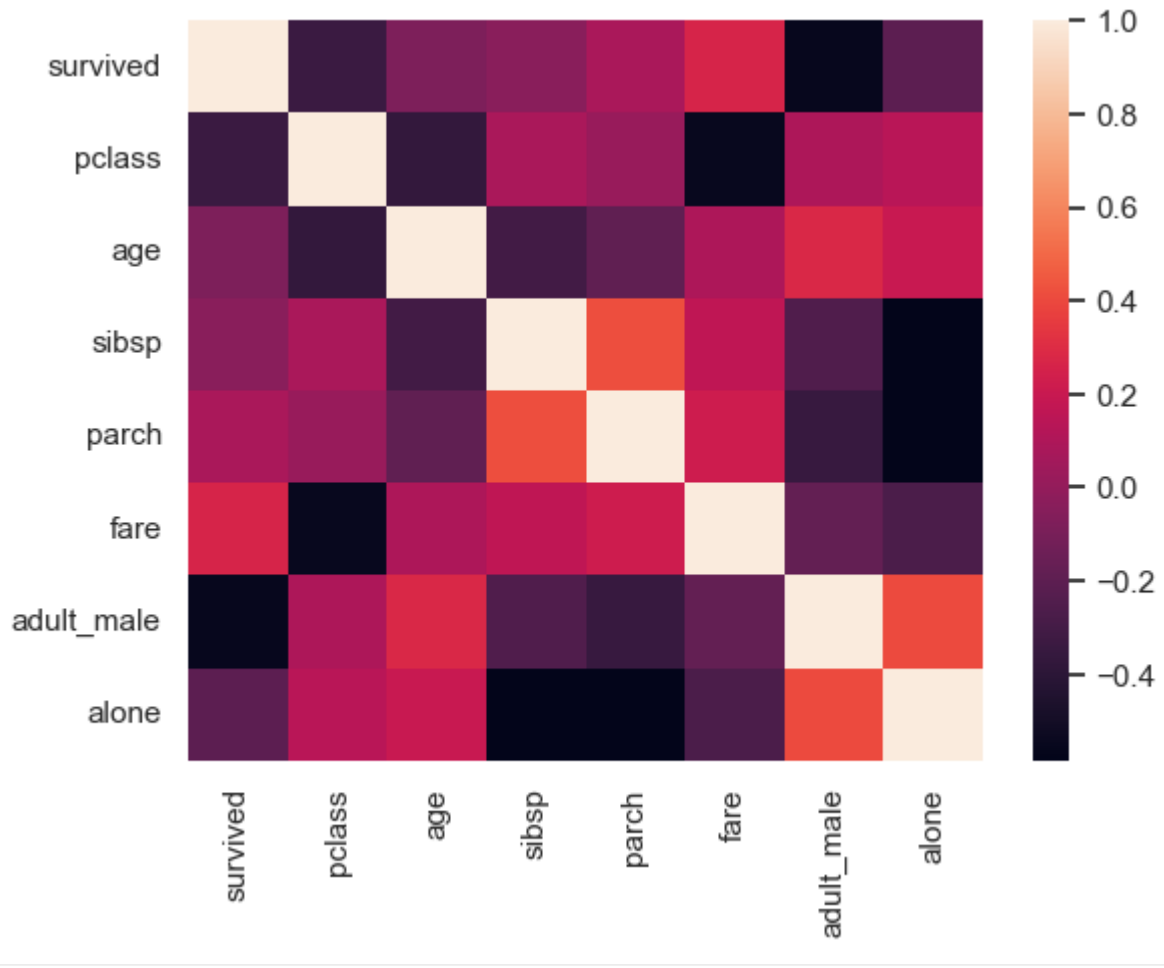
	survived	pclass	age	sibsp	parch	fare	adult_male	alone
survived	1.000000	-0.338481	-0.077221	-0.035322	0.081629	0.257307	-0.557080	-0.203367
pclass	-0.338481	1.000000	-0.369226	0.083081	0.018443	-0.549500	0.094035	0.135207
age	-0.077221	-0.369226	1.000000	-0.308247	-0.189119	0.096067	0.280328	0.198270
sibsp	-0.035322	0.083081	-0.308247	1.000000	0.414838	0.159651	-0.253586	-0.584471
parch	0.081629	0.018443	-0.189119	0.414838	1.000000	0.216225	-0.349943	-0.583398
fare	0.257307	-0.549500	0.096067	0.159651	0.216225	1.000000	-0.182024	-0.271832
adult_male	-0.557080	0.094035	0.280328	-0.253586	-0.349943	-0.182024	1.000000	0.404744
alone	-0.203367	0.135207	0.198270	-0.584471	-0.583398	-0.271832	0.404744	1.000000

(continues on next page)

(continued from previous page)

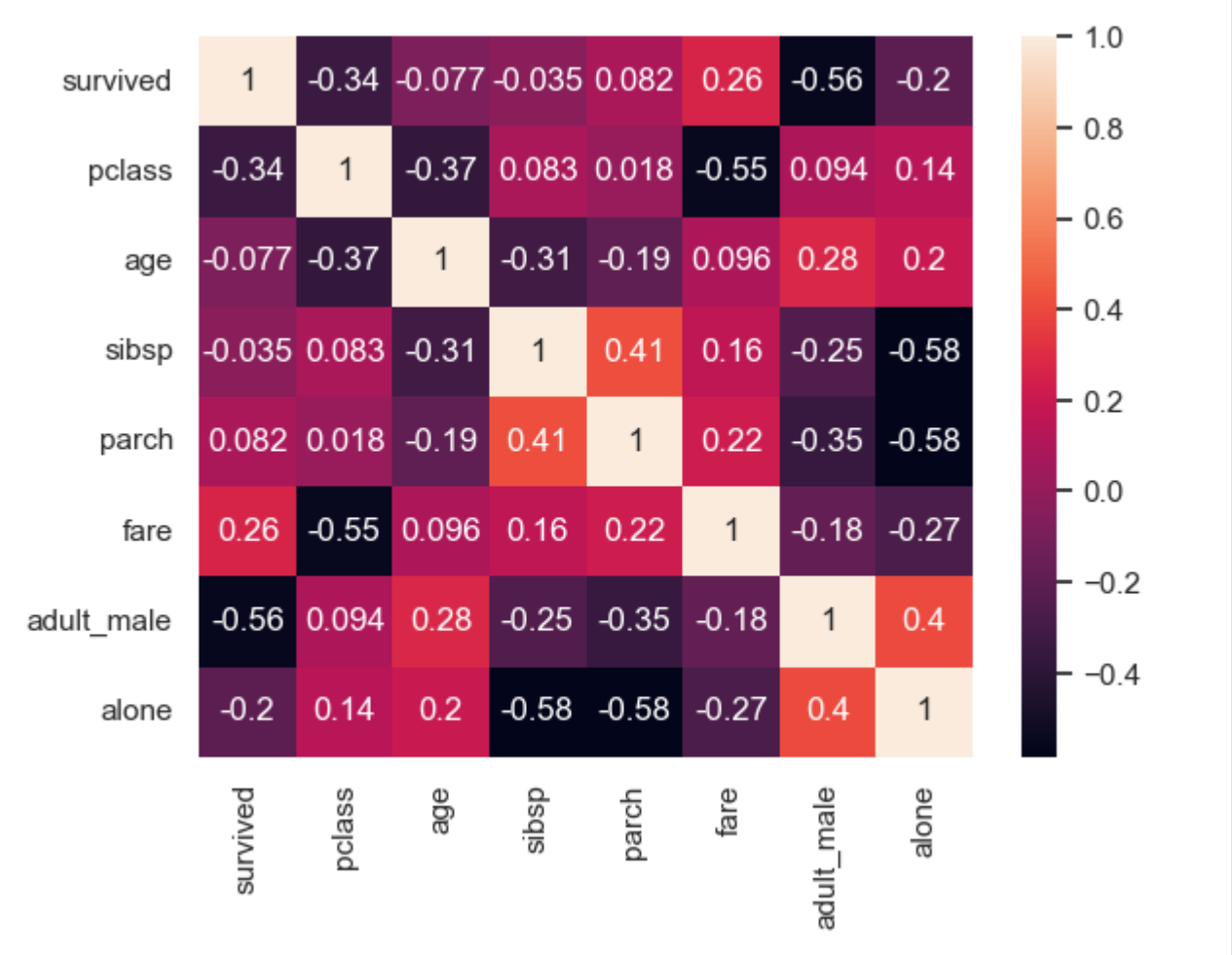
```
age           0.280328  0.198270
sibsp        -0.253586 -0.584471
parch        -0.349943 -0.583398
fare         -0.182024 -0.271832
adult_male    1.000000  0.404744
alone         0.404744  1.000000
```

[66]:
```
sns.heatmap(correlation);
```



If we want to add the values to the heat map, we can set the parameter `annot` to True.

[67]:
```
sns.heatmap(correlation, annot=True);
```



7.12.7 12.7 Styles, Themes, and Colors

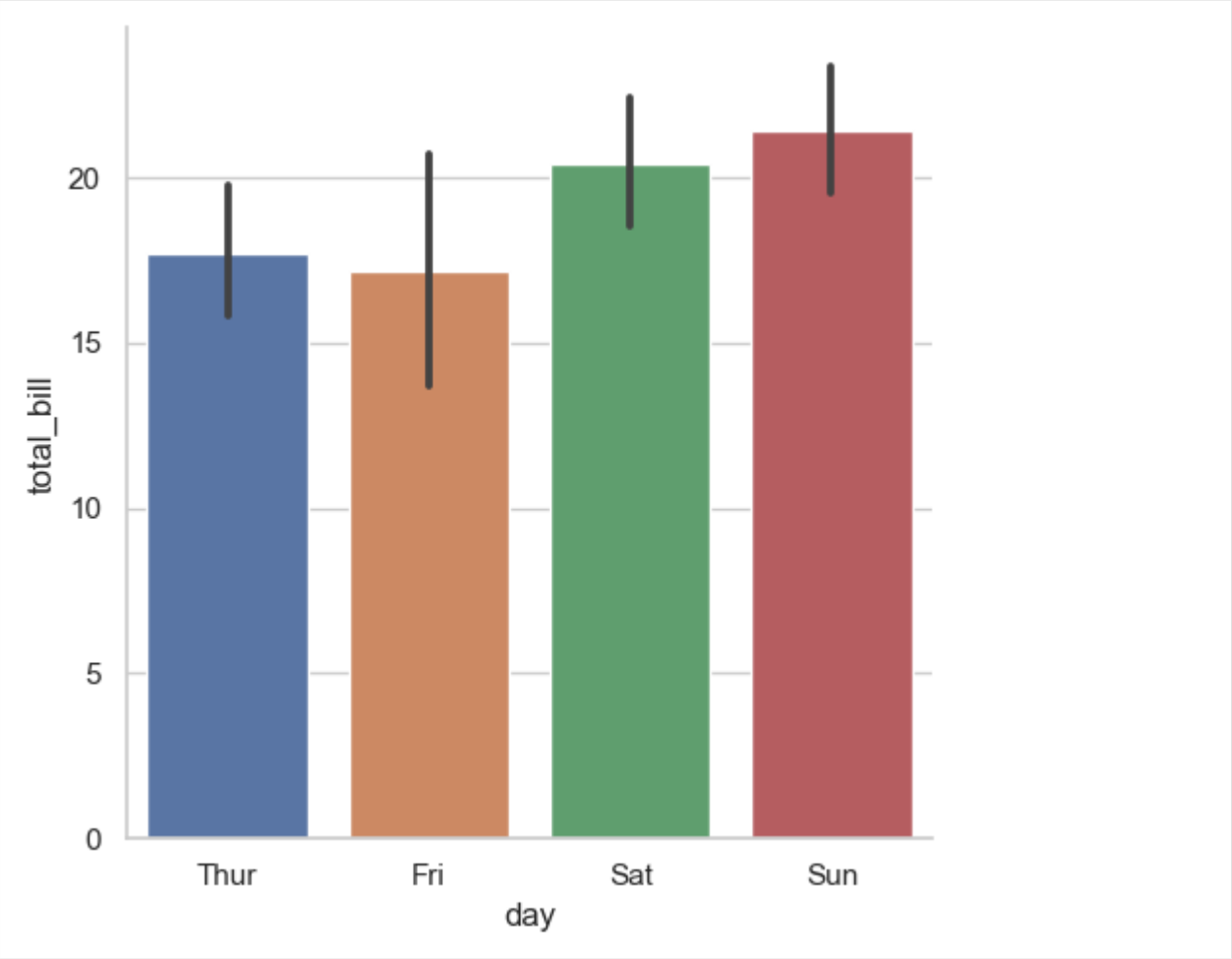
Seaborn allows to customize the visualizations depending on our needs, and provides ways to control the styles, themes, and colors in our plots.

Styles and Themes

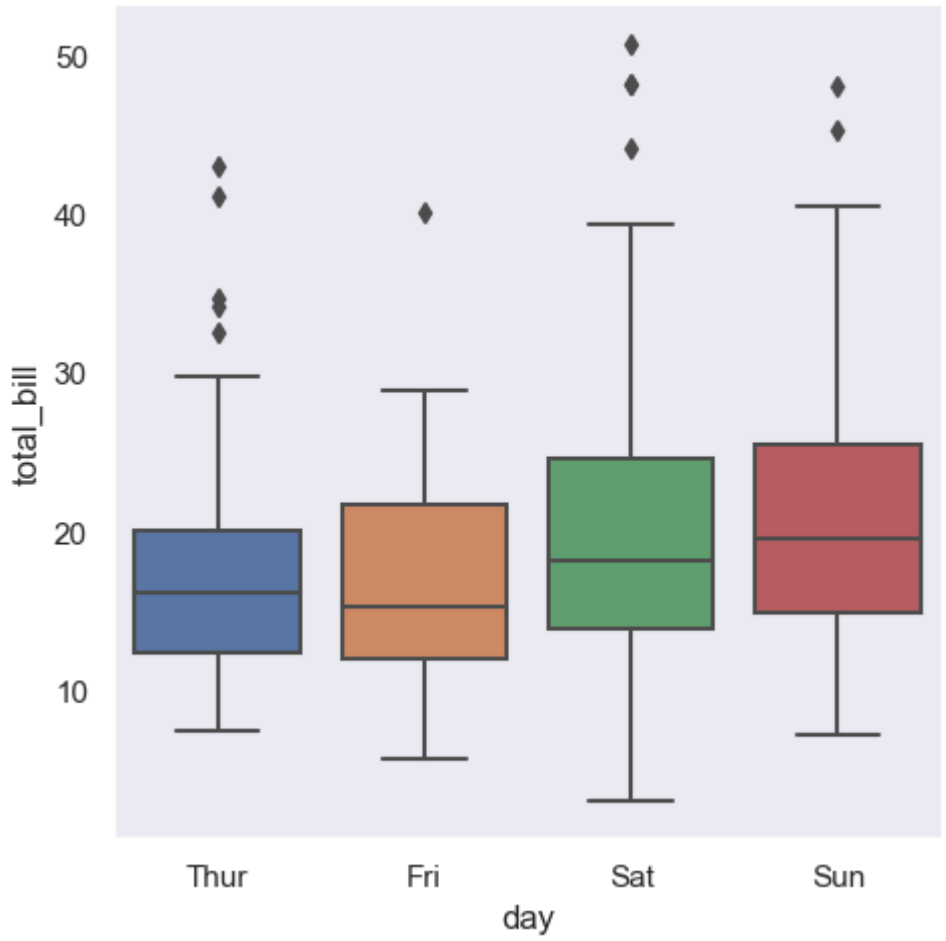
There are five styles in Seaborn plots:

- `darkgrid` (default) provides a dark background with grid lines
- `whitegrid` provides a white background with grid lines
- `dark` provides a dark background without grid lines
- `white` provides a white background without grid lines
- `ticks` provides a light gray background with tick marks on the axes

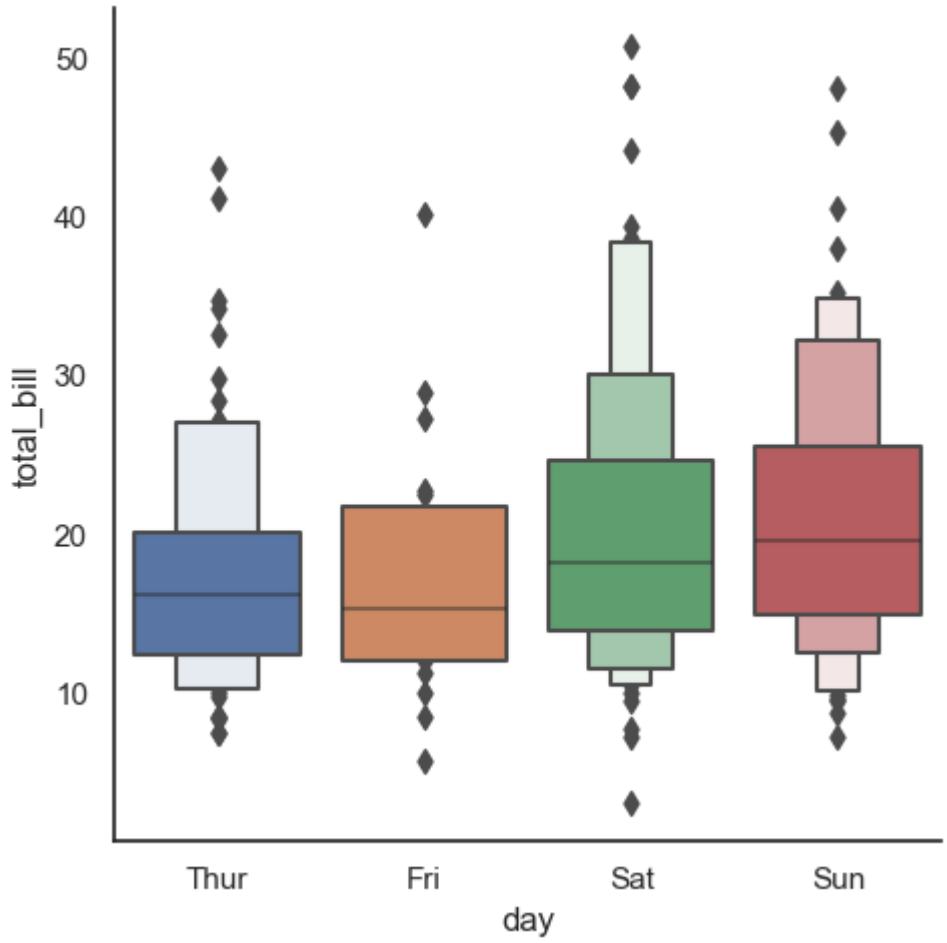
[68]:
```
sns.set_style('whitegrid')
sns.catplot(data=tips, x='day', y='total_bill', kind='bar');
```



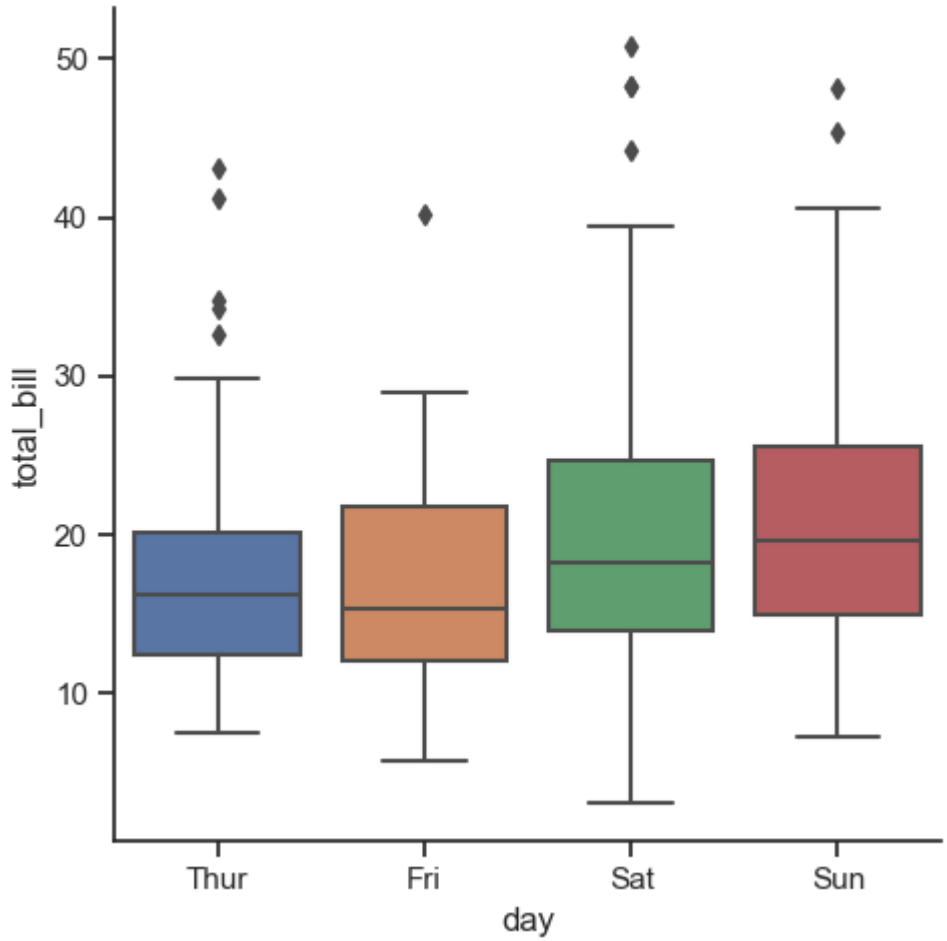
[69]:
```
sns.set_style('dark')
sns.catplot(data=tips, x='day', y='total_bill', kind='box');
```



[70]:
```
sns.set_style('white')
sns.catplot(data=tips, x='day', y='total_bill', kind='boxen');
```



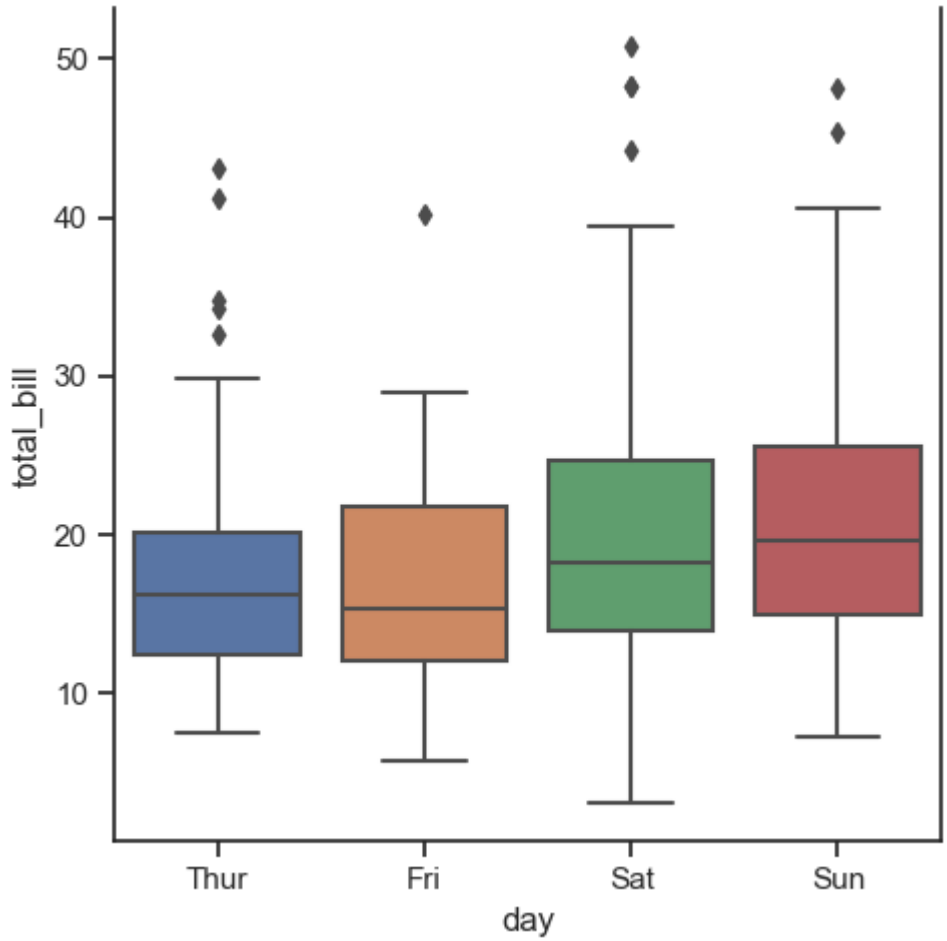
[71]:
```
sns.set_style('ticks')
sns.catplot(data=tips, x='day', y='total_bill', kind='box');
```



Removing the Axes Spines

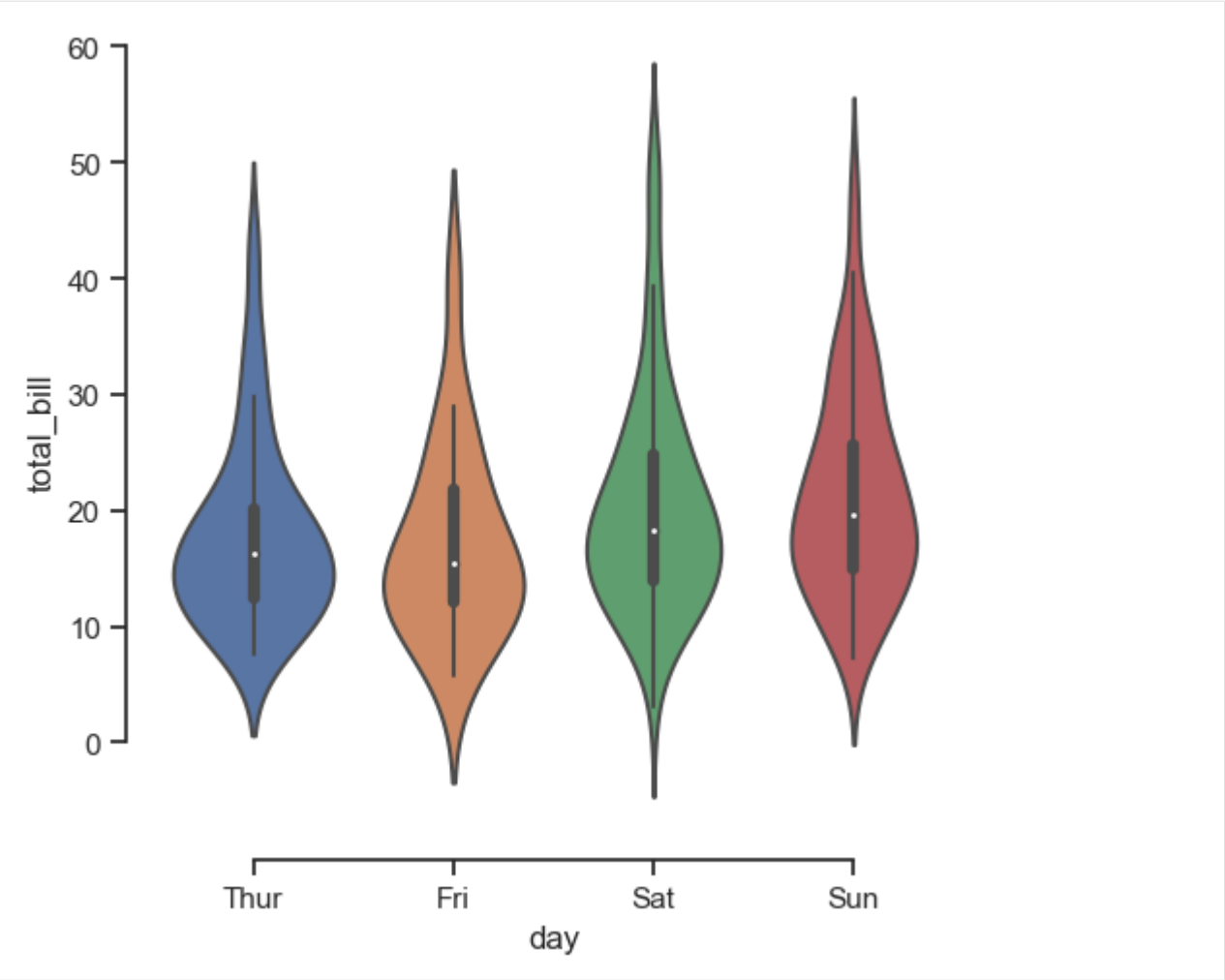
We can use the function `despine()` to remove the axes spines (borders) in the plots, or to show the spines by setting them to False.

[72]:
```
sns.catplot(data=tips, x='day', y='total_bill', kind='box')
sns.despine(right=False);
```



We can also move the spines away from the data by setting the `offset` distance that spines should move away from the axes.

[73]:
```
sns.catplot(data=tips, x='day', y='total_bill', kind='violin')
sns.despine(offset=10, trim=True);
```



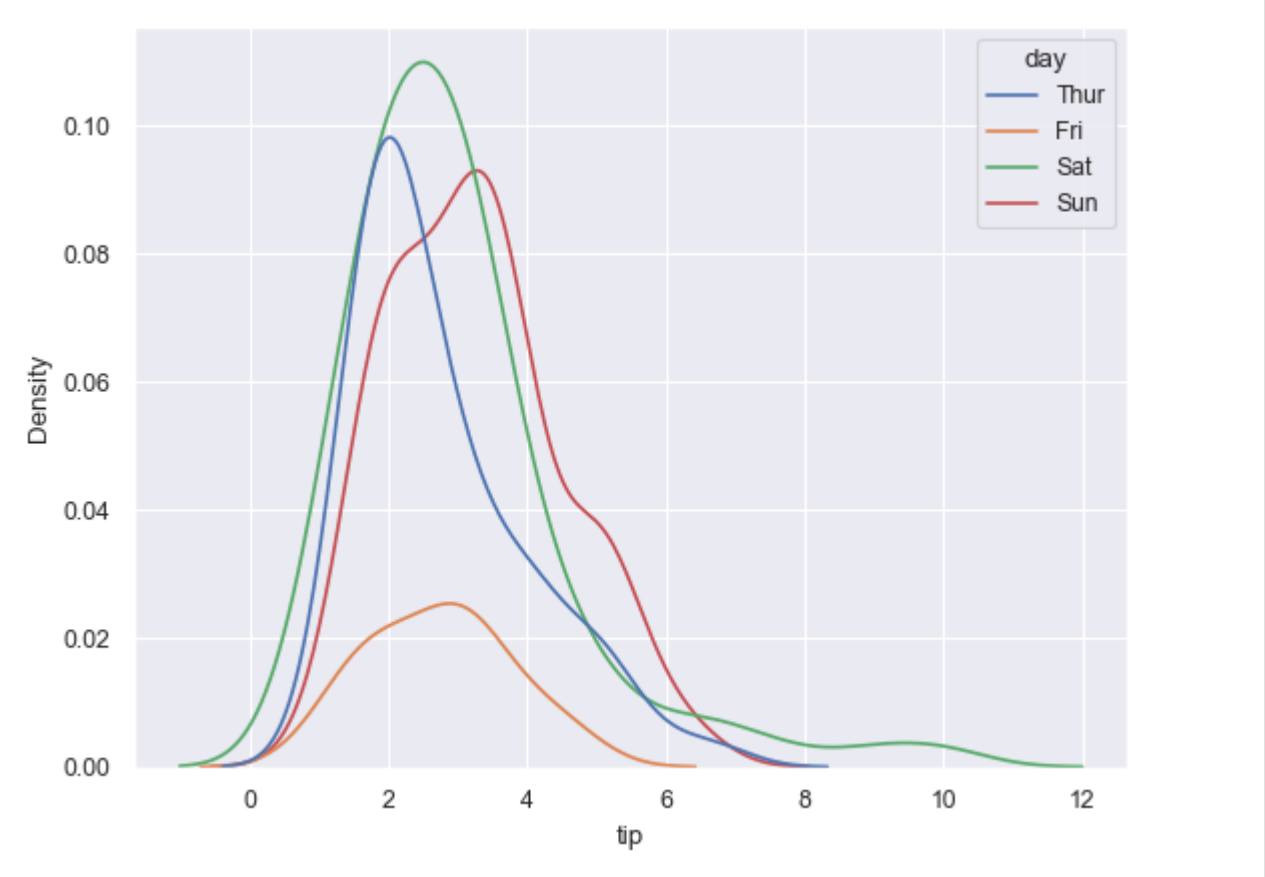
Scaling Plot Elements with Context

Context is used to control the scale of the elements of the plot. This can be helpful depending on where we want to use the visualizations.

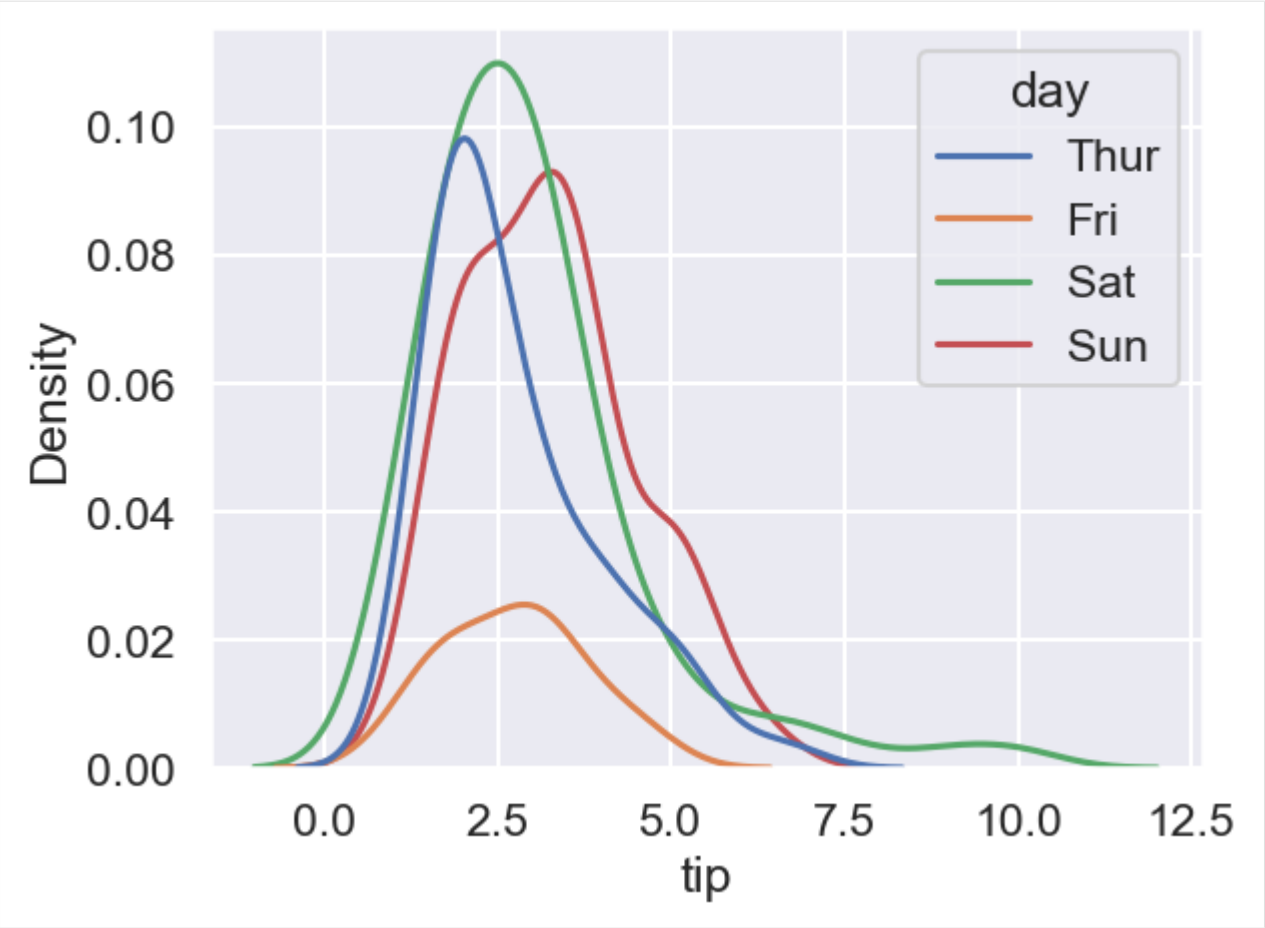
The function `set_context()` allows to select four contexts: `paper`, `notebook` (default), `talk`, and `poster`.

[74]:
```
sns.set_theme()
```

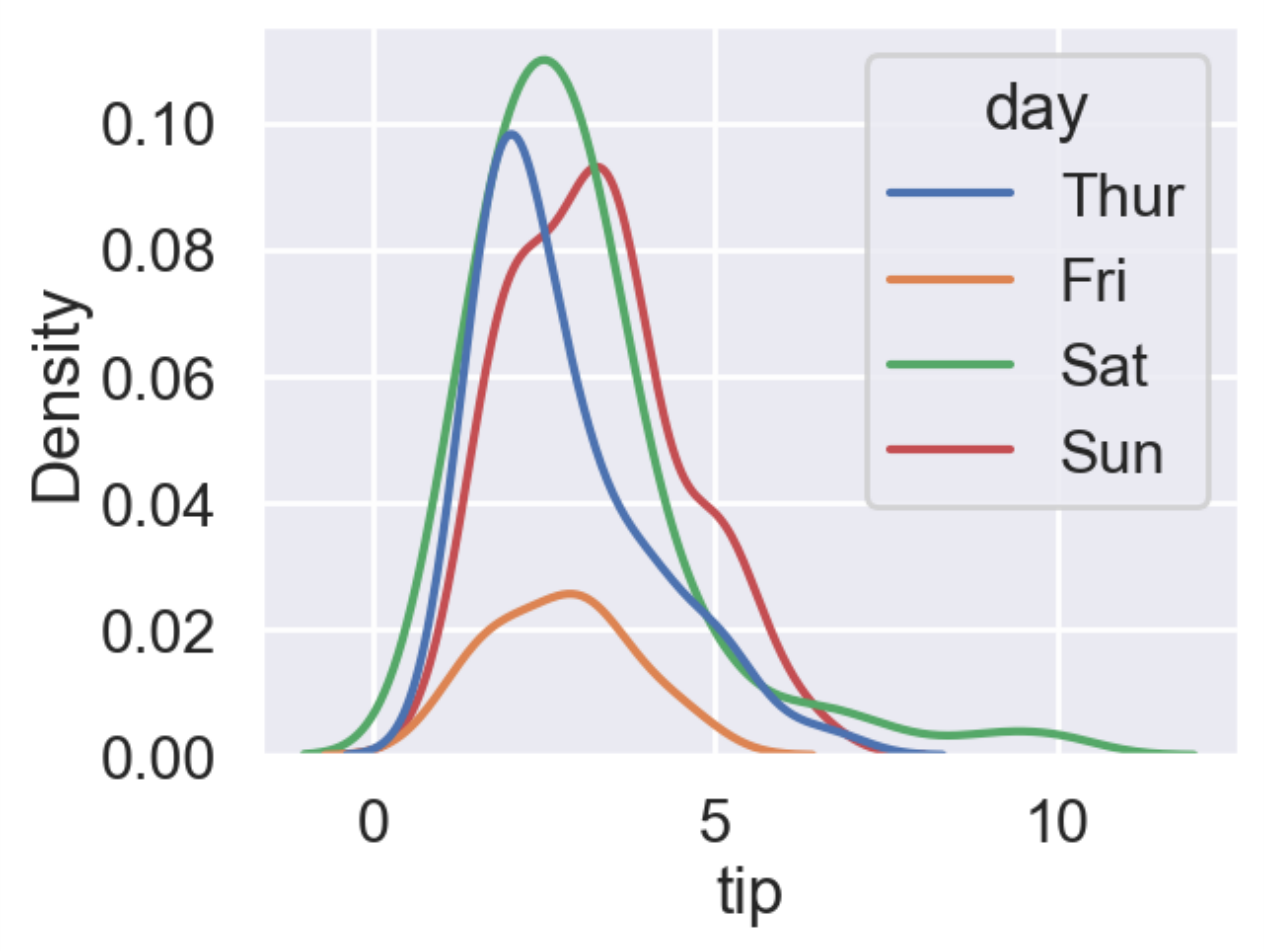
[75]:
```
sns.set_context('paper')
sns.kdeplot(data=tips, x='tip', hue='day');
```



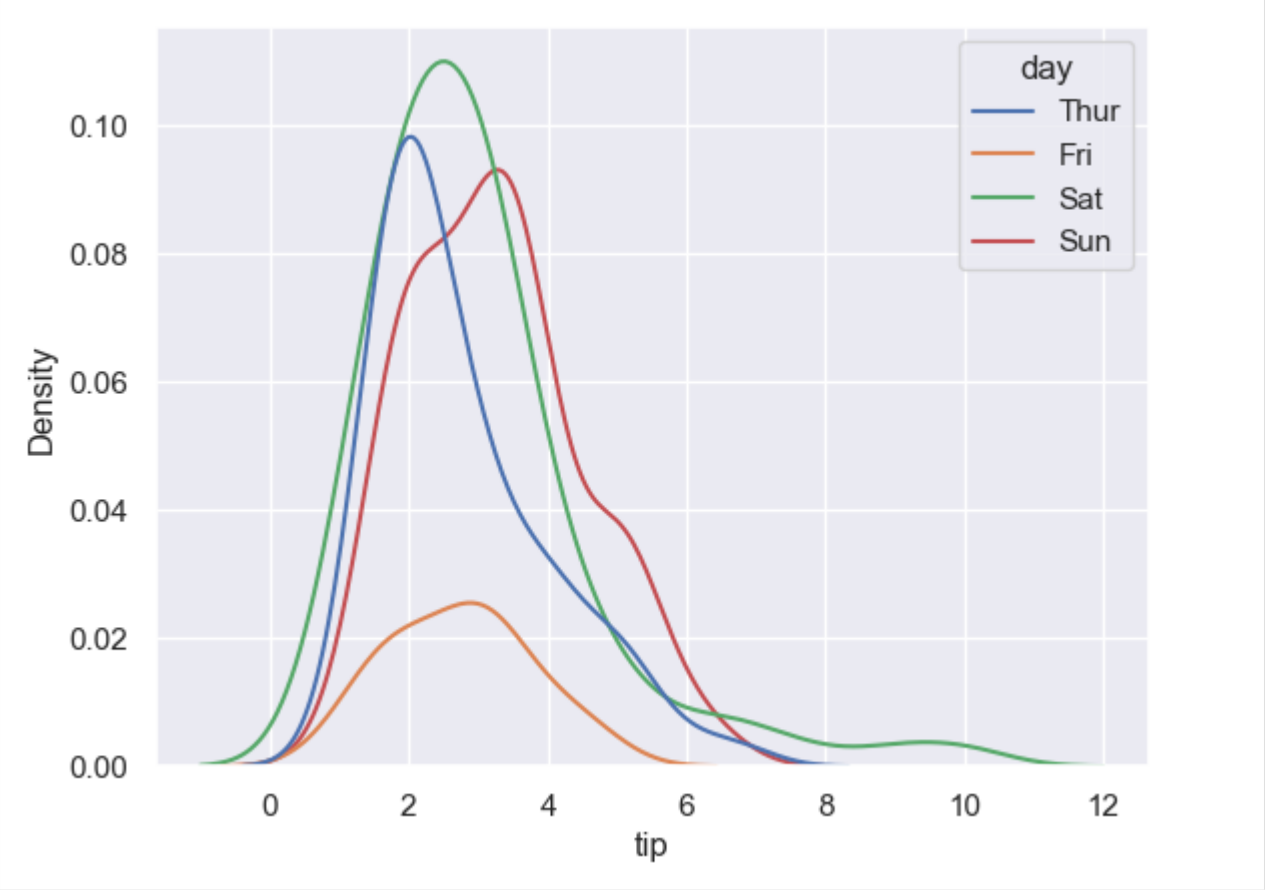
[76]:
```
sns.set_context('talk')
sns.kdeplot(data=tips, x='tip', hue='day');
```



[77]:
```
sns.set_context('poster')
sns.kdeplot(data=tips, x='tip', hue='day');
```



[78]:
```
sns.set_context('notebook')
sns.kdeplot(data=tips, x='tip', hue='day');
```

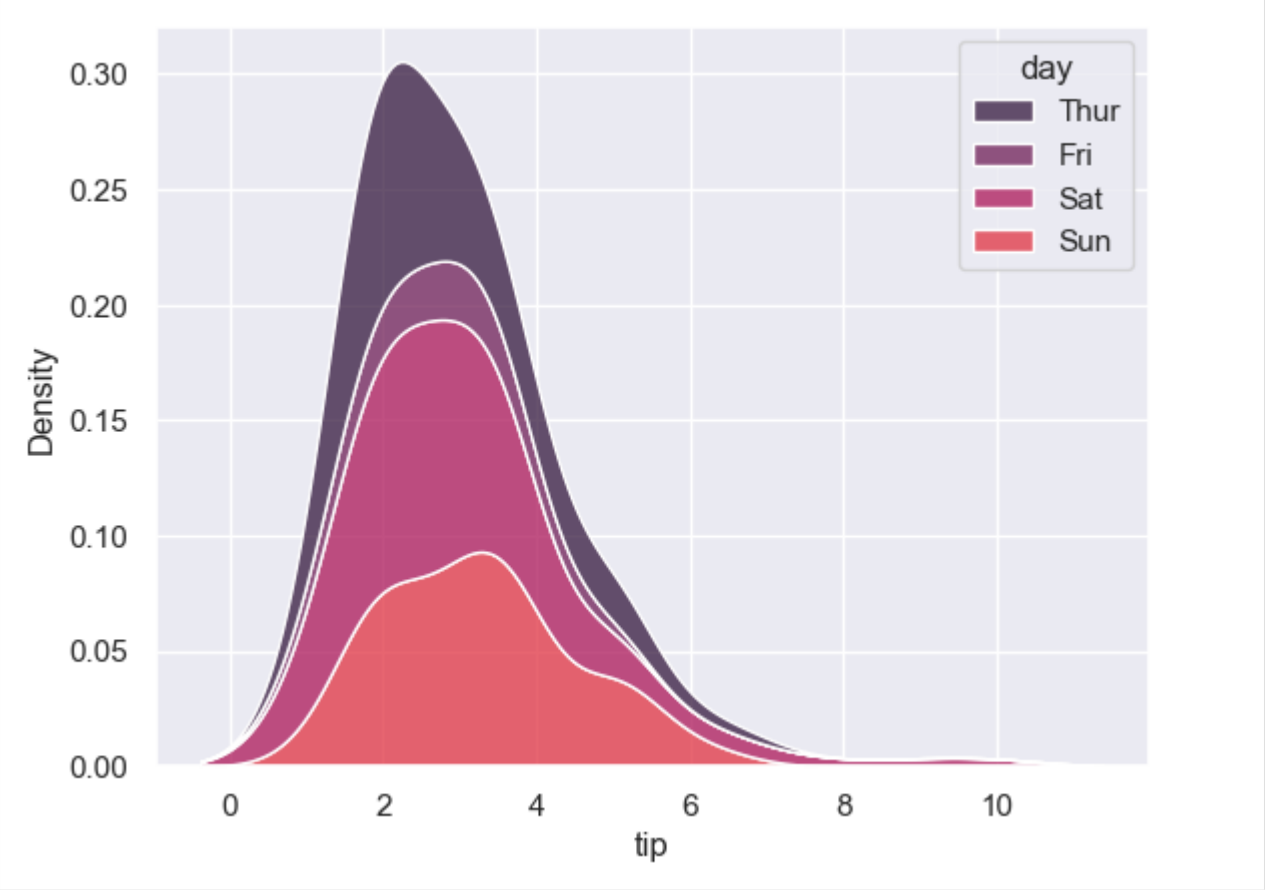


Colors

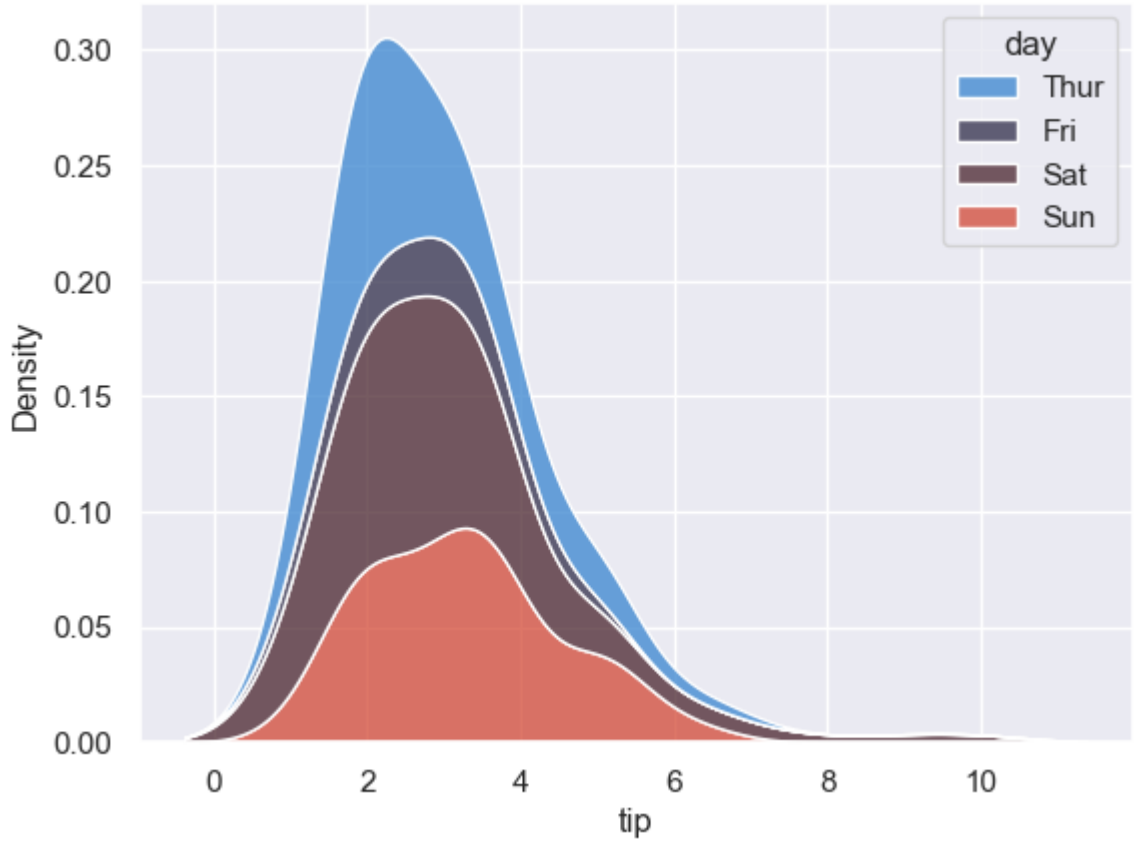
Seaborn allows us to choose colors from a wide range of color palettes for plot visualization.

To specify preferred colors, we can either use `color_palette()` before each plot, or set the parameter `palette` inside the plot definition.

[79]:
```
sns.set_palette('rocket')
sns.kdeplot(data=tips, x='tip', hue='day', multiple='stack');
```



[80]:
```
sns.kdeplot(data=tips, x='tip', hue='day', multiple='stack', palette='icefire');
```



To see the available color palettes, check out the Seaborn documentation.

[81]:
```
# Check the palettes

sns.color_palette('tab10')
```

[81]:
```
[(0.1215686274509039, 0.4666666666666667, 0.7058823529411765),
 (1.0, 0.4980392156862745, 0.054901960784313725),
 (0.17254901960784313, 0.6274509803921569, 0.17254901960784313),
 (0.8392156862745098, 0.15294117647058825, 0.1568627450980392),
 (0.5803921568627451, 0.403921568627451, 0.7411764705882353),
 (0.5490196078431373, 0.33725490196078434, 0.29411764705882354),
 (0.8901960784313725, 0.4666666666666667, 0.7607843137254902),
 (0.4980392156862745, 0.4980392156862745, 0.4980392156862745),
 (0.7372549019607844, 0.7411764705882353, 0.13333333333333333),
 (0.09019607843137255, 0.7450980392156863, 0.8117647058823529)]
```

[82]:
```
sns.color_palette('dark')
```

[82]:
```
[(0.0, 0.10980392156862745, 0.4980392156862745),
 (0.6941176470588235, 0.25098039215686274, 0.050980392156862744),
 (0.07058823529411765, 0.44313725490196076, 0.10980392156862745),
 (0.5490196078431373, 0.03137254901960784, 0.0),
 (0.34901960784313724, 0.11764705882352941, 0.44313725490196076),
 (0.34901960784313724, 0.1843137254901961, 0.050980392156862744),
```

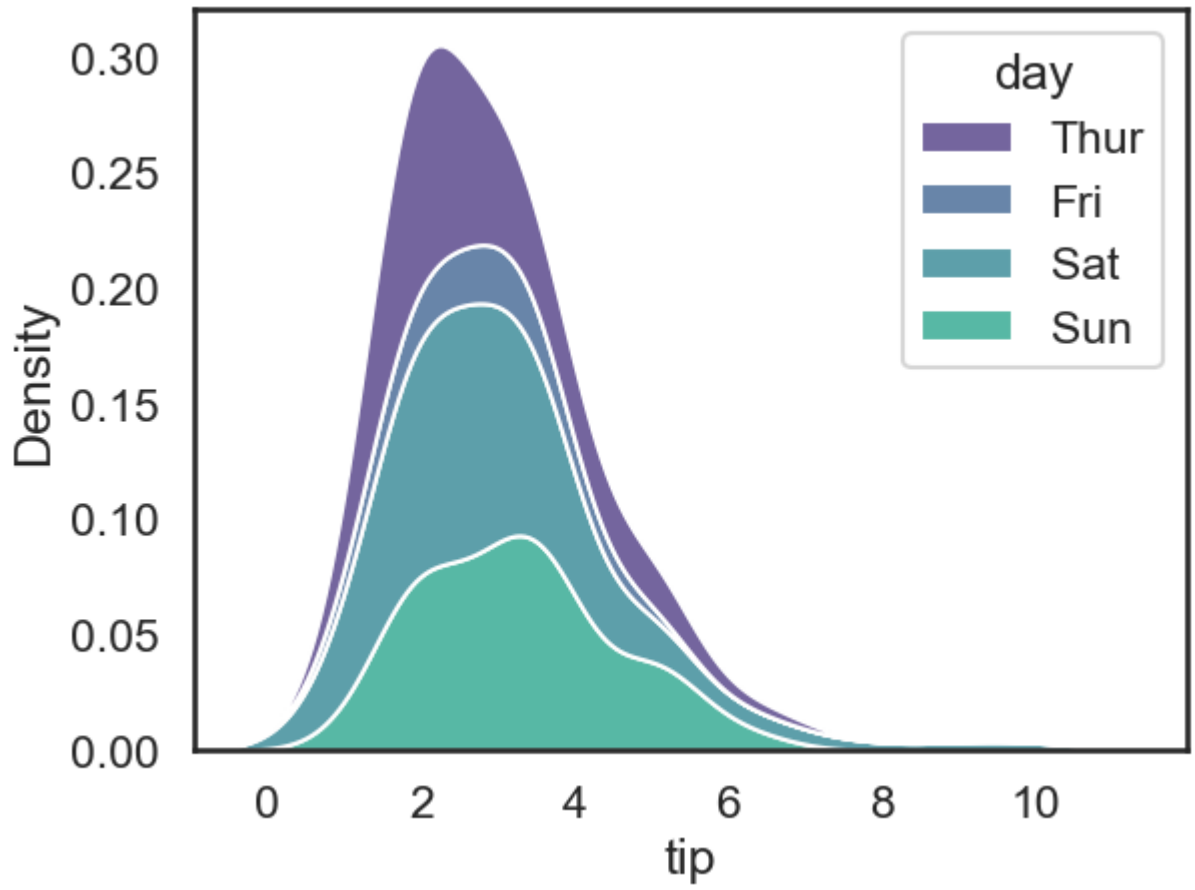
(continues on next page)

(continued from previous page)

```
(0.6352941176470588, 0.20784313725490197, 0.5098039215686274),
(0.23529411764705882, 0.23529411764705882, 0.23529411764705882),
(0.7215686274509804, 0.5215686274509804, 0.0392156862745098),
(0.0, 0.38823529411764707, 0.4549019607843137)]
```

Also, we can use `sns.set_theme()` to directly set the style, palette, and context.

[83]:
```
sns.set_theme(style='white', context='talk', palette='viridis')
sns.kdeplot(data=tips, x='tip', hue='day', multiple='stack');
```



7.12.8 References

1. Complete Machine Learning Package, Jean de Dieu Nyandwi, available at: https://github.com/Nyandwi/machine_learning_complete.
2. Seaborn Tutorial: An Introduction to Seaborn, available at: https://seaborn.pydata.org/tutorial/introduction.html.

BACK TO TOP

7.13 Lecture 13 - Scikit-Learn Library for Data Science

7.13.1 13.1 Introduction to Scikit-Learn

Scikit-Learn is a Python library designed to provide access to machine learning algorithms within Python code, through a clean and well-designed API. It has been built by hundreds of contributors from around the world, and is used in industry and academia.

Scikit-Learn is built upon Python's NumPy (Numerical Python) and SciPy (Scientific Python) libraries, which enable efficient numerical and scientific computation within Python.

Scikit-Learn is imported via the `sklearn` module.

[1]:
```
import sklearn
```

Representation of Data in Scikit-Learn

Most machine learning algorithms implemented in scikit-learn expect the training data to be stored in a **two-dimensional array** (matrix). The size of the array is expected to be `[n_samples, n_features]`.

- **n_samples** is the number of samples (also referred to as inputs, data points, instances, examples), where each sample is an item to process (e.g., classify). A sample can be a row in database or CSV file, an image, a video, a document, an astronomical object, or anything that we can describe with a set of quantitative features.
- **n_features** is the number of features or distinct characteristics that can be used to describe each sample in a quantitative manner. Features are generally real-valued numbers, but they can also be boolean or discrete values.

A Dataset Example: Iris Dataset

As an example of a simple dataset, we are going to look at the `Iris` dataset stored by the scikit-learn library.

The data consists of measurements of three different species of irises:

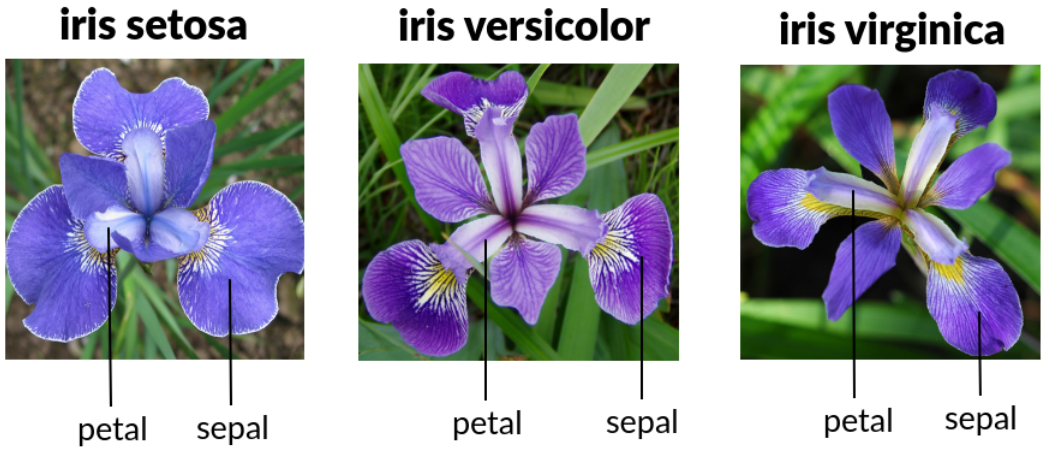
1. Iris Setosa
2. Iris Versicolour
3. Iris Virginica

Each sample has 4 features, which include:

1. Sepal length in cm
2. Sepal width in cm
3. Petal length in cm
4. Petal width in cm

All four features are measured in centimeters (cm). Note that the data are not images of iris flowers, but 4 measurements for each flower.

Examples of images from the three iris species are shown below.



There are 50 samples for each species, therefore there are 150 samples in total for all 3 species.

Scikit-learn provides a helper function `load_iris` to load the dataset into a dictionary with numpy arrays as values.

[2]:
```
from sklearn.datasets import load_iris
iris = load_iris()
```

Let's explore the keys in the loaded dictionary. The main elements are the `data` and the `target` keys.

[3]:
```
iris.keys()
```

[3]:
```
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'filename
↪', 'data_module'])
```

[4]:
```
# 150 samples, 4 features
iris.data.shape
```

[4]:
```
(150, 4)
```

[5]:
```
# Show the features for the first sample
iris.data[0]
```

[5]:
```
array([5.1, 3.5, 1.4, 0.2])
```

[6]:
```
# 'Target' is the array of class labels for each sample
iris.target.shape
```

[6]:
```
(150,)
```

[7]:
```
print(iris.target)
# 0 is Iris Setosa, 1 is Iris Versicolor, 2 is Iris Virginica
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

[8]:
```
print(iris.target_names)
```

```
['setosa' 'versicolor' 'virginica']
```

[9]:
```
print(iris.feature_names)
```

```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

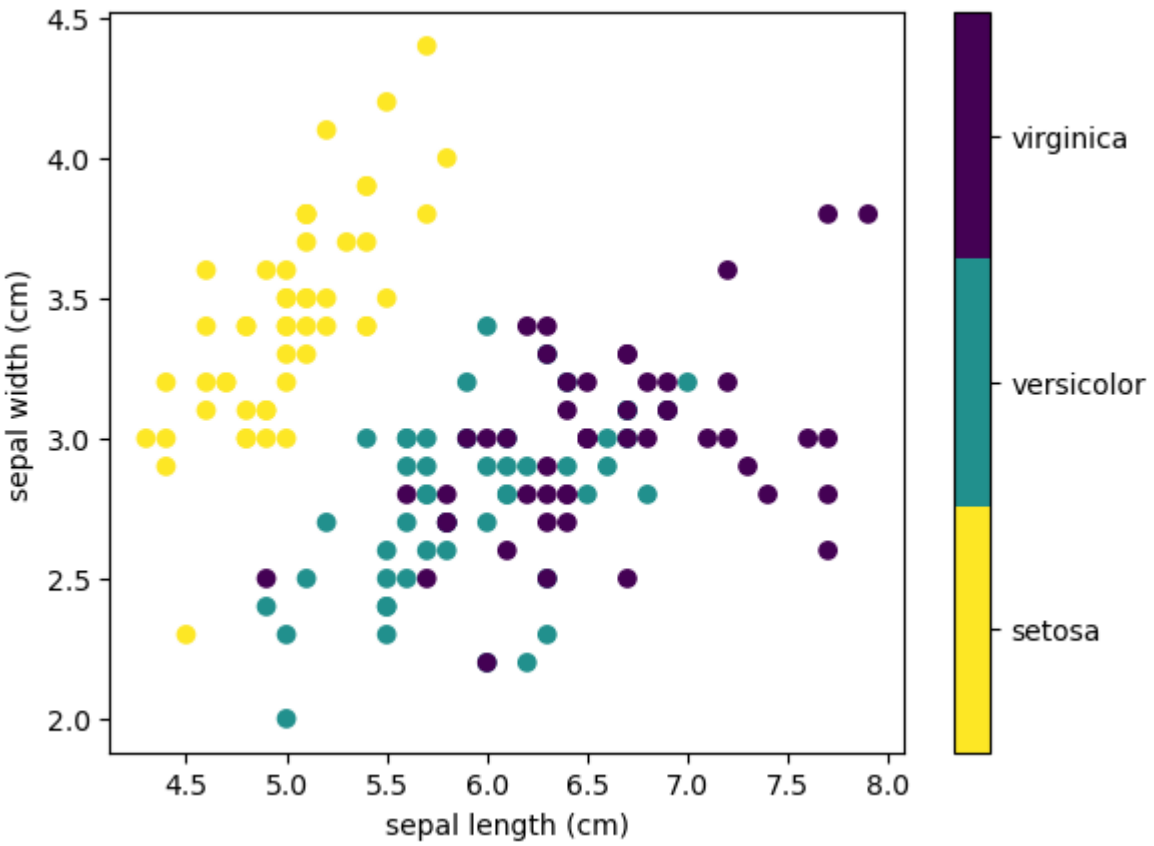
Although the dataset is four-dimensional, we can visualize two of the dimensions at a time using a scatter plot.

[10]:
```
import numpy as np
import matplotlib.pyplot as plt

import warnings
warnings.filterwarnings('ignore')

x_index = 0    # sepal_length
y_index = 1    # sepal_width

# this formatter will label the colorbar with the correct target names
formatter = plt.FuncFormatter(lambda i, *args: iris.target_names[int(i)])
```

(continues on next page)

(continued from previous page)

```
plt.scatter(iris.data[:, x_index], iris.data[:, y_index],
            c=iris.target, cmap=plt.cm.get_cmap('viridis_r', 3))
plt.colorbar(ticks=[0, 1, 2], format=formatter)
plt.clim(-0.5, 2.5)
plt.xlabel(iris.feature_names[x_index])
plt.ylabel(iris.feature_names[y_index]);
```



Other Available Data in Scikit-Learn

Besides the Iris dataset, scikit-learn provides other datasets, that can be loaded directly from scikit-learn. The datasets include:

- **Packaged Data:** several small datasets are packaged with the scikit-learn installation, and can be downloaded using the tools in `sklearn.datasets.load_*`
- **Downloadable Data:** several larger datasets are available for download, and scikit-learn includes tools that streamline this process. These tools can be found in `sklearn.datasets.fetch_*`
- **Generated Data:** several datasets can be generated using various random sample generators. These are available in `sklearn.datasets.make_*`

[11]:
```
from sklearn import datasets
```

To see the available datasets in scikit-learn use the following:

```
# Type datasets.load_<TAB> to see all available datasets
datasets.load_

# Type datasets.fetch_<TAB> to see all available datasets
datasets.fetch_

# Type datasets.make_<TAB> to see all available datasets
datasets.make_
```

7.13.2 13.2 Supervised Learning: Classification

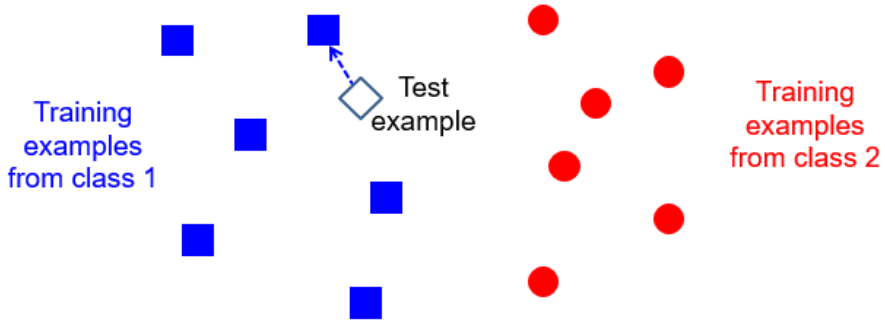
In **supervised learning**, an algorithm processes a dataset consisting of samples (training data) and labels. Supervised learning is further broken down into two main categories: **classification** and **regression**. In classification, the label is discrete such as the class membership, while in regression the label is continuous. We will first look at examples of performing classification with scikit-learn, and afterwards we will study regression tasks.

The implementations of machine learning algorithms and models in scikit-learn are called **estimators** or estimator objects. Each estimator can be fitted to input data using the `fit()` method in scikit-learn. Fitting an estimator to data is equivalent to training a model, that is, the parameters of the model are learned from the data.

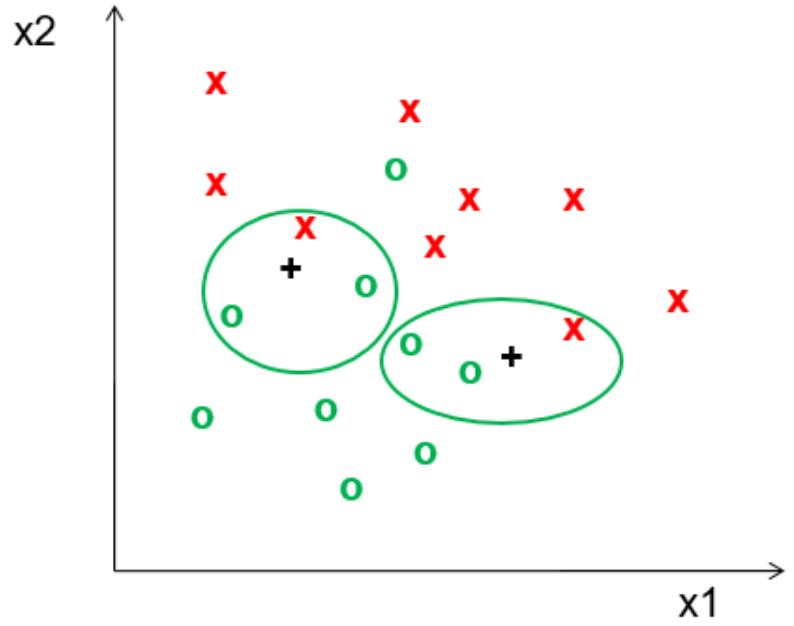
13.2.1 k-Nearest Neighbors (kNN)

We will illustrate classification first with the **k-nearest neighbors (kNN)** algorithm. This algorithm uses a very simple learning strategy: to predict the target class of a new sample, it takes into account its k closest samples in the training set and predicts the class labels based on majority voting of these samples.

For instance, if k=1, to assign a class label to the test example shown with the white square in the figure below, the kNN classifier will first calculate the distance to each data point in the training set, and afterward, it will assign the class of the nearest data point to the test example.



Or, if k=3, then the 3 nearest neighbors will be considered to classify a test data point. For the test data points shown with the + marker in the figure below, the class is obtained by voting based on the class labels of the 3 closest data points.



kNN is a simple classification algorithm, and one main disadvantage is that the classifier needs to remember all of the training data and store it for future comparisons with test samples. This can be space inefficient when working with large datasets having many data points, and it can be computationally expensive since it requires calculating the distance to all data points.

Let's try kNN on the iris classification problem. In the following cell, we passed the newly created classifier object to the name `knn` and we chose to use 5 nearest neighbors.

[12]:
```
from sklearn import neighbors

# create the model
knn = neighbors.KNeighborsClassifier(n_neighbors=5)

X, y = iris.data, iris.target
```

The full set of arguments that can be passed to kNN in scikit-learn is shown below, and more information about the arguments can be found at this link.

```
class sklearn.neighbors.KNeighborsClassifier(n_neighbors=5, *, weights='uniform',
algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None)
```

Train the Model

[13]:
```
# fit the model with model.fit(training data, training target)
knn.fit(X, y)
```

[13]:
```
KNeighborsClassifier()
```

The model training is depicted in the following diagram. The model is fit to the training data/samples/instances (often denoted `X`) and training target/label/class/category (often denoted `y`). The method `fit` uses the specified **learning algorithm** for estimating the parameters of the model. The learned parameters constitute the **model state** that can later be used to make predictions (to classify test data).

Note again the syntax for training/fitting a model: `model.fit(data, target)`

Figure source: Reference [5].

Make Predictions

Next, we will use our trained model to make predictions on the same dataset `X` that we used for training. To predict, the model uses the input data together with the model state to output a target/label for each data point.

Note the syntax for predicting with a model: `model.predict(data)`

Figure source: Reference [5].

[14]:
```
y_predicted = knn.predict(X)

# y_predicted is an array of predicted class labels for each input
y_predicted
```

[14]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 2, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

[15]:
```
y_predicted.shape
```

[15]:
```
(150,)
```

The output of `model.predict()` is an array with a class label assigned to each data instance. The array `y_predicted` has 150 items, because we passed an array `X` with 150 data instances.

Compare the predicted class labels to the ground-truth target array `y` given below.

[16]:
```
# Ground-truth target array
y
```

[16]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

We can also make a prediction only on a subset of the data. For example, the next cell shows the prediction for the first two samples. The predicted class labels are 0 and 0.

[17]:
```
result = knn.predict(X[0:2])
```

[18]:
```
result
```

[18]:
```
array([0, 0])
```

Note that `predict` expects the data to be two-dimensional, and if we input only 1 sample, we will get an error.

[19]:
```
result_1 = knn.predict(X[0])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16588\255161176.py in <module>
----> 1 result_1 = knn.predict(X[0])

~\anaconda3\lib\site-packages\sklearn\neighbors\_classification.py in predict(self, X)
    212             Class labels for each data sample.
    213         """
--> 214         neigh_dist, neigh_ind = self.kneighbors(X)
    215         classes_ = self.classes_
    216         _y = self._y

~\anaconda3\lib\site-packages\sklearn\neighbors\_base.py in kneighbors(self, X, n_
↪neighbors, return_distance)
    715                 X = _check_precomputed(X)
    716             else:
--> 717                 X = self._validate_data(X, accept_sparse="csr", reset=False)
    718         else:
    719             query_is_train = True

~\anaconda3\lib\site-packages\sklearn\base.py in _validate_data(self, X, y, reset,␣
↪validate_separately, **check_params)
    564             raise ValueError("Validation should be done on X, y or both.")
    565         elif not no_val_X and no_val_y:
--> 566             X = check_array(X, **check_params)
    567             out = X
    568         elif no_val_X and not no_val_y:

~\anaconda3\lib\site-packages\sklearn\utils\validation.py in check_array(array, accept_
↪sparse, accept_large_sparse, dtype, order, copy, force_all_finite, ensure_2d, allow_nd,
↪ ensure_min_samples, ensure_min_features, estimator)
    767             # If input is 1D raise error
    768             if array.ndim == 1:
--> 769                 raise ValueError(
    770                     "Expected 2D array, got 1D array instead:\narray={}.\n"
    771                     "Reshape your data either using array.reshape(-1, 1) if "

ValueError: Expected 2D array, got 1D array instead:
array=[5.1 3.5 1.4 0.2].
Reshape your data either using array.reshape(-1, 1) if your data has a single feature or␣
↪array.reshape(1, -1) if it contains a single sample.
```

This is because the shape of the first sample is `(4,)`. If we make the sample two-dimensional with `X[np.newaxis, 0]`, then we can perform a prediction without error.

[20]:
```
X[0].shape
```

[20]:
```
(4,)
```

[21]:
```
# This works, we changed the shape from (4,) to (1,4)
result_1 = knn.predict(X[np.newaxis, 0])
```

[22]:
```
print(result_1)
```
```
[0]
```

Or, we can use the suggestion in the error message, and use `array.reshape(1,-1)`.

[23]:
```
first_sample = X[0].reshape(1,-1)
first_sample.shape
```
[23]:
```
(1, 4)
```

[24]:
```
result_2 = knn.predict(first_sample)
print(result_2)
```
```
[0]
```

Similarly, we can make predictions by specifying arbitrary input features; for example, an iris flower that has 3cm x 5cm sepal and 4cm x 2cm petal, as follows.

[25]:
```
# what kind of iris has 3cm x 5cm sepal and 4cm x 2cm petal?
result_2 = knn.predict([[3, 5, 4, 2]])
```

[26]:
```
print(result_2)
```
```
[1]
```

[27]:
```
print(iris.target_names[result_2])
```
```
['versicolor']
```

We can also perform probabilistic predictions with `predict_proba`, which outputs the probability that a sample belongs to each of the three iris species. In the output, the first element 1 means target class 0.

[28]:
```
knn.predict_proba(first_sample)
```
[28]:
```
array([[1., 0., 0.]])
```

Applying the `argmax()` function will output the index of the greatest element in the array.

[29]:
```
knn.predict_proba(first_sample).argmax()
```
[29]:
```
0
```

[30]:
```
# for the first 4 data samples
knn.predict_proba(X[0:4])
```
[30]:
```
array([[1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.]])
```

[31]:
```
knn.predict_proba(X[0:4]).argmax(axis=1)
```
[31]:
```
array([0, 0, 0, 0], dtype=int64)
```

Evaluate the Performance

Model evaluation (model validation) is determining how well the model predictions are relative to the known target labels.

To assess the accuracy of the prediction, we can compute the average success rate, i.e., all cases where the predicted target label `y_predicted` is the same as the ground-truth target label `y`.

[32]:
```
(y == y_predicted).mean()
```

[32]:
```
0.9666666666666667
```

Or, using `.sum().` we can see that the model predicted correctly 145 samples out of the 150 samples in the dataset `X`. The above function `mean()` just calculates the average as 145/150.

[33]:
```
(y == y_predicted).sum()
```

[33]:
```
145
```

And, instead of manually calculating the accuracy, we can use `model.score()` to calculate the average success rate. This method first performs prediction on a set of data inputs, and afterwards it calculates the accuracy of the predicted class labels in comparison to the true class labels.

The syntax is: `model.score(data, target)`

Figure source: Reference [5].

[34]:
```
accuracy = knn.score(X, y)
accuracy
```

[34]:
```
0.9666666666666667
```

One additional way to calculate the average success rate is to import the `accuracy_score` metric, and then apply it to find the match between ground-truth and predicted class labels.

[35]:
```
from sklearn.metrics import accuracy_score
accuracy_score(y, y_predicted)
```

[35]:
```
0.9666666666666667
```

Another useful way to examine the prediction results is to view the **confusion matrix**, that is, the matrix showing the frequency of inputs and outputs for each class. In the output of this cell, we can see in the first row that all 50 data points in class 0 (iris setosa) were correct. In the second row, we see that 47 data points in class 1 (iris versicolor) were correct, but 3 data points were predicted in class 2 (iris virginica). Similarly, 2 data points from class 2 were incorrectly classified as class 1. Apparently, there is some confusion between the second and third species. And, as we know, out of 150 data points, in total 5 data points were misclassified, which can be seen in the off-diagonal elements in the confusion matrix.

[36]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y, y_predicted))
```

```
[[50  0  0]
 [ 0 47  3]
 [ 0  2 48]]
```

Train-test Data Split

When building a machine learning model, it is important to evaluate the trained model on data that was not used to fit it, since the **generalization** ability of the model on new unseen data is of primary concern.

Correct evaluation of a model is done by leaving out a subset of the data when training the model, and using it afterwards for model evaluation.

Scikit-learn provides the function `sklearn.model_selection.train_test_split` to automatically split a dataset into two subsets: **training dataset** and **testing dataset**. This function shuffles the data and randomly splits it into a train and test set. Setting the `random_state` parameter allows to obtain deterministic results when using a random number generator. Setting `stratify=y` will ensure that the training and testing datasets have the same proportions of data points from the 3 classes of irises.

[37]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=123, test_size=0.
↪25, stratify=y)
```

[38]:
```
print('Training data inputs', X_train.shape)
print('Training labels', y_train.shape)
print('Testing data inputs', X_test.shape)
print('Testing labels', y_test.shape)
```

```
Training data inputs (112, 4)
Training labels (112,)
Testing data inputs (38, 4)
Testing labels (38,)
```

We can use the `bincount()` function to examine how the data from different classes were distributed in the training and testing datasets. In the next cell, we can see that the datasets are stratified.

[39]:
```
print(np.bincount(y_train))
print(np.bincount(y_test))
```

```
[38 37 37]
[12 13 13]
```

Let's re-train and re-evaluate the model.

[40]:
```
# 1. initialize the model
knn_model = neighbors.KNeighborsClassifier()

# 2. fit the model on train data
knn_model.fit(X_train, y_train)

# 3. evaluate the model on test data
accuracy = knn_model.score(X_test, y_test)

print(f'The test accuracy is {round(accuracy, 4) * 100}%')
```

```
The test accuracy is 97.37%
```

This example was not the best though, because the obtained score on the unseen test data (97.37%) is typically lower than the accuracy obtained by evaluating the model on the training data (96.67% in this case). This is probably due to the small dataset. In practice, evaluating a machine learning model on the training data is unacceptable in all scenarios.

Finally, how do we know how many nearest neighbors to use: 5, 7, 29? Well, it is not easy to answer that question. First, we can make an intuitive guess based on our understanding of the dataset, or second, we can run the model several times using different numbers of nearest neighbors and see which model performs the best.

Plot the Decision Boundary

If we select only 2 features of the iris data, we can also inspect the decision boundary. Let's select the last two features, fit a model, and plot the decision boundary for the model using the function in the following cell.

[41]:
```
X1 = iris.data[:, [2, 3]] # select only the last two features
y1 = iris.target

X1_train, X1_test, y1_train, y1_test = train_test_split(X1, y1, random_state=123, test_
↪size=0.25, stratify=y1)

knn_model_2 = neighbors.KNeighborsClassifier()

knn_model_2.fit(X1_train, y1_train)

accuracy = knn_model_2.score(X1_test, y1_test)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))
```

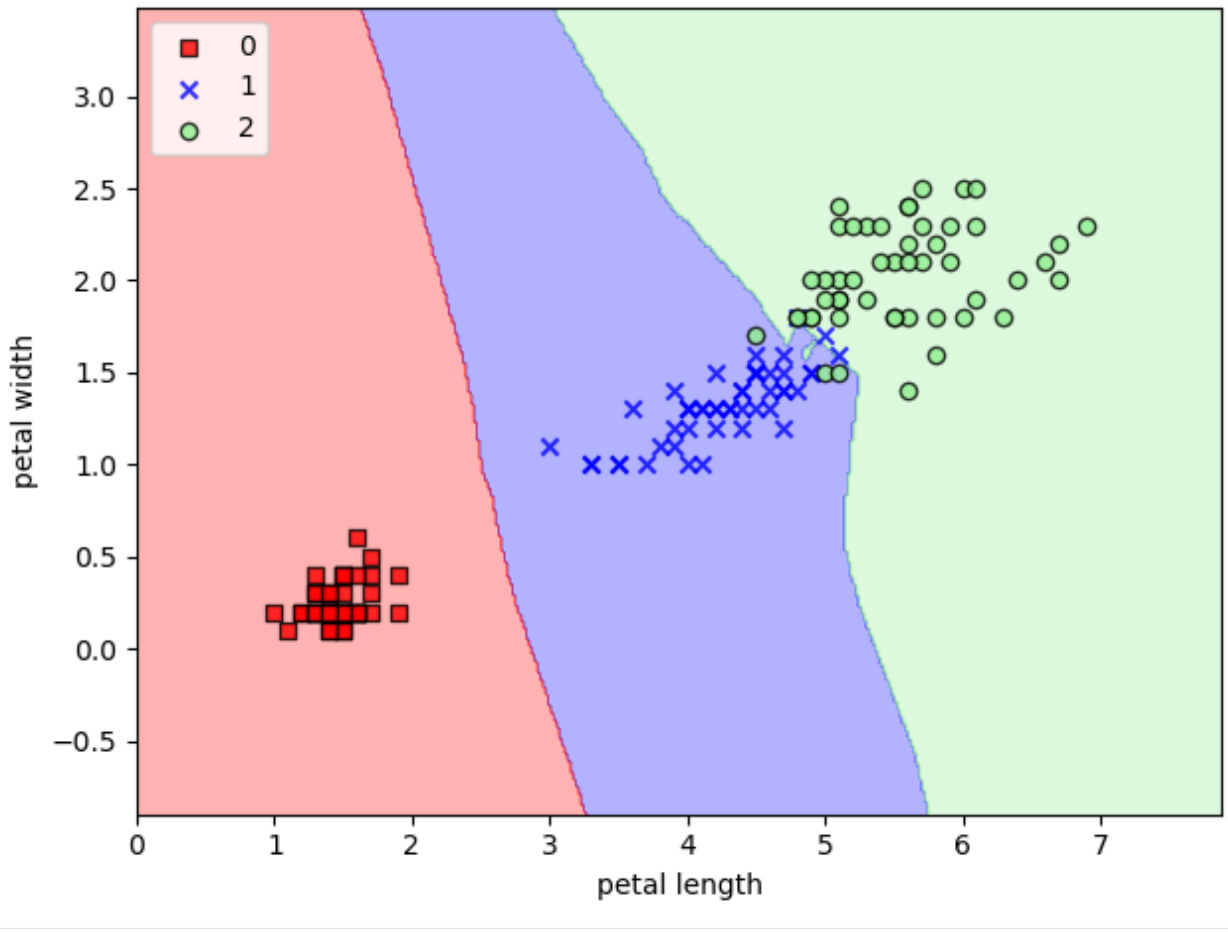
```
The test accuracy is 97.37 %
```

[42]:
```
# The code in this cell is not required to quizzes or assignments

from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):

    # Adapted from Python Machine Learning (2nd Ed.) Code Repository, Sebastian Raschka,
    # available at: [https://github.com/rasbt/python-machine-learning-book-2nd-edition]

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
```

(continues on next page)

(continued from previous page)

```
                    c=colors[idx],
                    marker=markers[idx],
                    label=cl,
                    edgecolor='black')

    plt.xlabel('petal length')
    plt.ylabel('petal width')
    plt.legend(loc='upper left')
    plt.tight_layout()
    plt.show()

plot_decision_regions(X1, y1, classifier=knn_model_2)
```

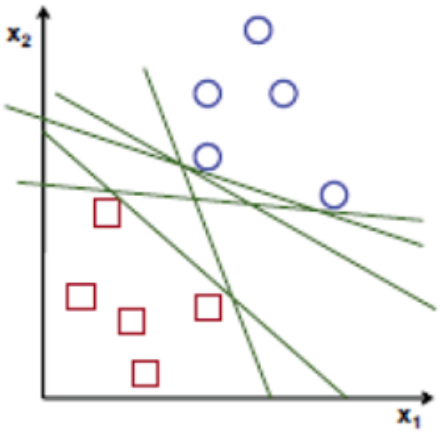


13.2.2 Support Vector Machines (SVM)

Support Vector Machines are supervised learning algorithms used for classification, regression, and outlier detection. SVM are one of the most powerful models in machine learning suited for handling complex and high-dimensional datasets.

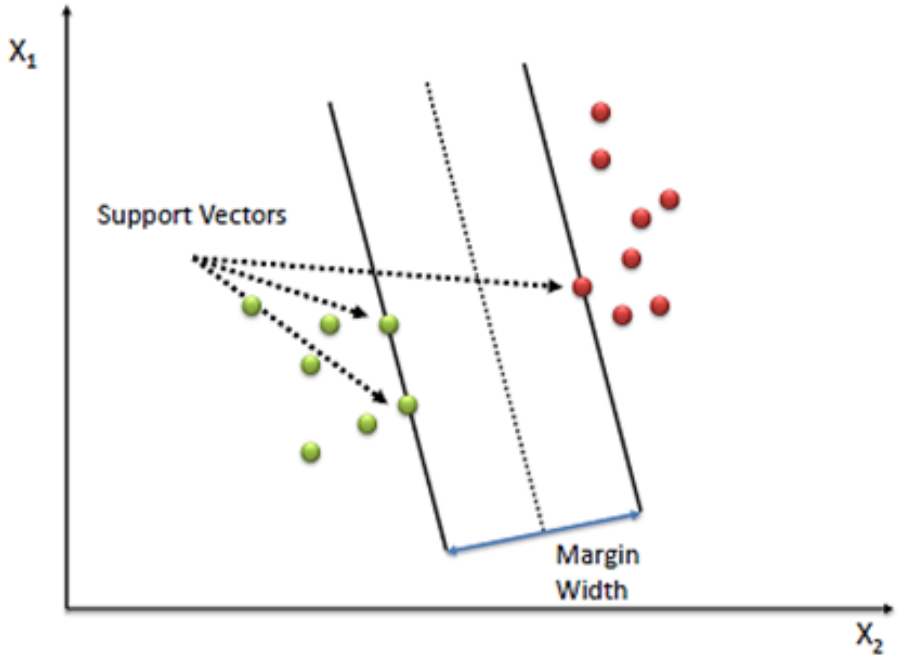
SVM algorithm solves an optimization problem to identify a *decision boundary* (also referred to as *hyperplane*) that separates class instances in an optimal manner. Specifically, depending on the dimensionality of the data, to separate two-dimensional data into two groups we can use a line (one dimension), then to separate three-dimensional data we can use a plane (two dimensions), or to separate N-dimensional data we can use a hyperplane (N-1 dimensions).

For instance, in the following figure, all lines correctly separate the classes of blue circles and red squares. The question is how to find the best decision boundary?



SVM solves this problem by finding the maximum *margin* between the decision boundary and the data points, where the main assumption is that the line that is farthest from all training examples will have better generalization capabilities. Conversely, hyperplanes that pass too close to the training examples will be sensitive to noise and, therefore, less likely to generalize well for data outside the training set.

The algorithm first identifies a classifier that correctly classifies all the examples, and afterwards increases the geometric margin until it cannot increase the margin any further. In the following figure, the maximum margin boundary is specified by three data points, called *support vectors*.



An advantage of SVM is that if the data is linearly separable, there is a unique global solution. An ideal SVM analysis should produce a hyperplane that completely separates the data points into two non-overlapping classes. However, perfect separation may not be possible, and in that situation, SVM finds the hyperplane that maximizes the margin and minimizes the misclassifications.

The mathematical details regarding SVM and other machine learning models are omitted in this lecture. Instead, we will just treat the scikit-learn algorithm as a black box which fits an SVM and predicts on the data.

Let's import SVM classifier from scikit-learn and fit it to the iris dataset. Note the `SVC` stands for Support Vector Classifier.

[43]:
```
from sklearn.svm import LinearSVC

lin_svm = LinearSVC()
lin_svm.fit(X_train, y_train)
```

[43]:
```
LinearSVC()
```

[44]:
```
linsvm_pred = lin_svm.predict(X_test)

accuracy_score(y_test, linsvm_pred)
```

[44]:
```
1.0
```

[45]:
```
confusion_matrix(y_test, linsvm_pred)
```

[45]:
```
array([[12,  0,  0],
       [ 0, 13,  0],
       [ 0,  0, 13]], dtype=int64)
```

And recall again that we could have obtained the same result using the `score` function.

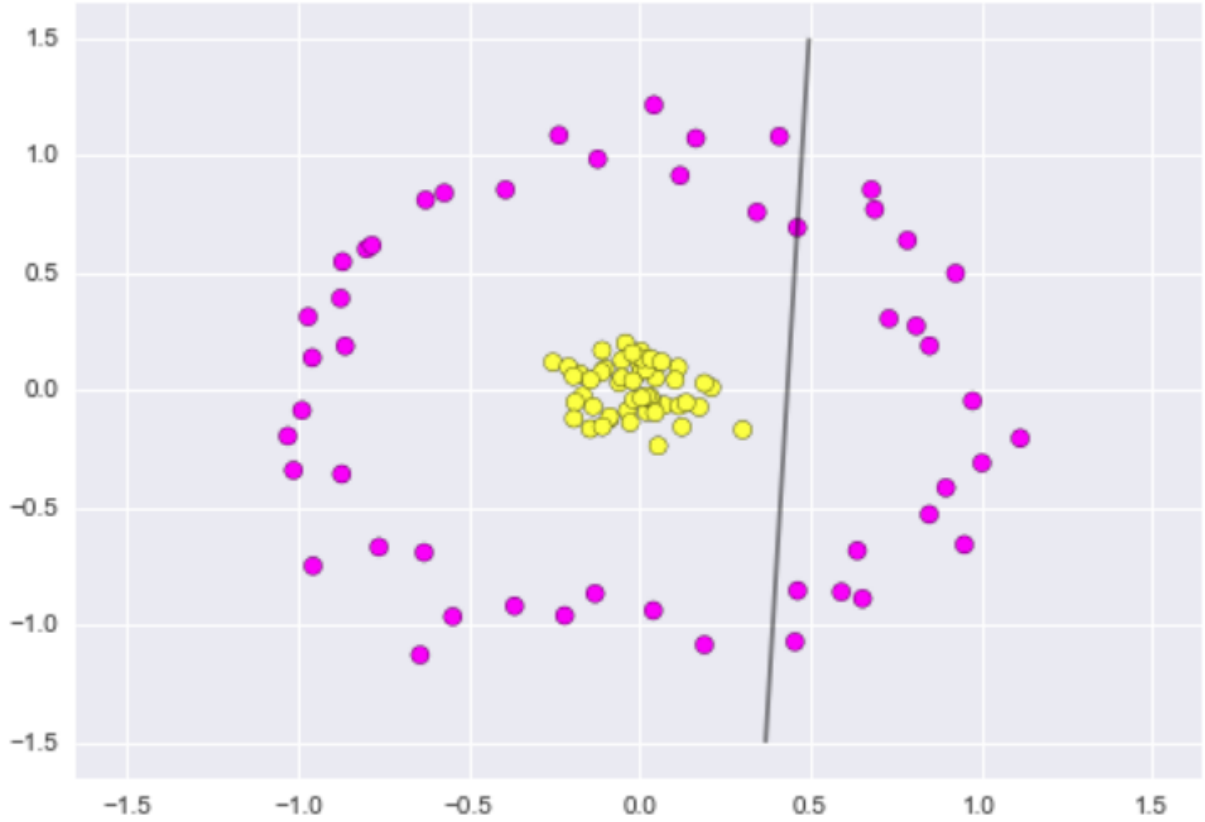
[46]:
```
lin_svm.score(X_test, y_test)
```

[46]:
```
1.0
```

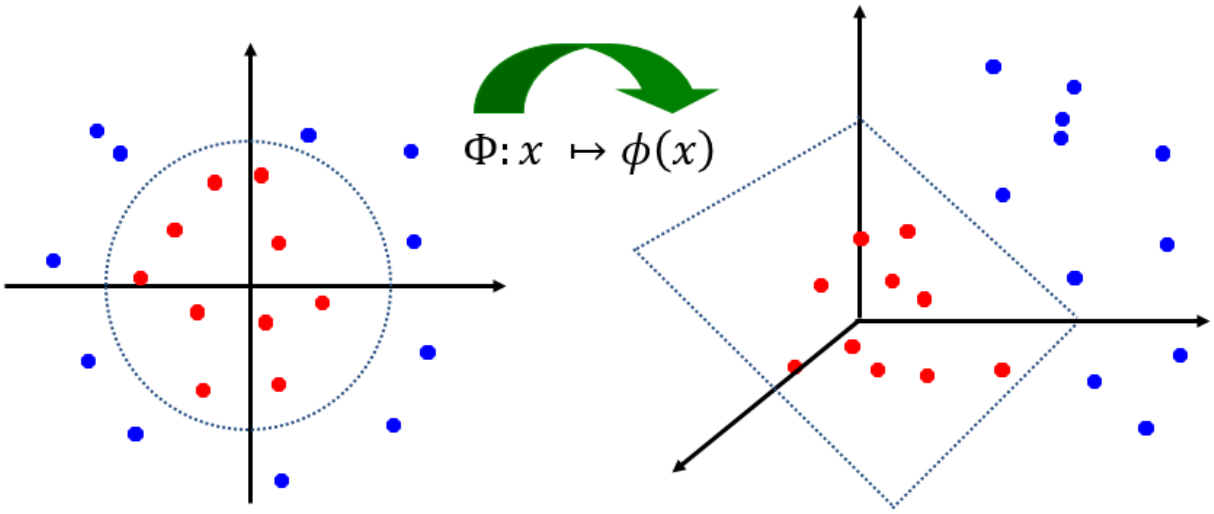
Interestingly, SVM achieved 100% accuracy on the iris dataset.

SVM Kernel Methods

Although the Linear SVM method is very efficient when the data points are linearly separable, in cases when the data are not linearly separable (as the data in the following figure), SVM will fail, as obviously, no linear discrimination can separate these data points.



SVM can handle such cases by using a **kernel function**, which applies a functional transformation of the input data into a different space. This is shown in the following figure, where a non-linear kernel function ϕ is used to map the data x into a different space where the data is linearly separable. This is called the *kernel trick*, which means the kernel function transforms the data into a higher dimensional feature space to make it possible to perform the linear separation.



SVM supports several kernels, including `linear`, `polynomial`, `sigmoid`, and `rbf` kernels. Linear kernel is just the regular SVM. Among the non-linear kernels, Gaussian radial basis function (RBF) kernel is one of the most commonly used. In scikit-learn, SVM with a linear kernel can be imported as LinearSVC (as we did above), while SVC provides access to the different kernels (meaning that LinearSVC can be implemented as SVC with the argument `kernel = 'linear'`).

Let's try to apply SVM with polynomial and rbf kernels to the Iris dataset and compare the results.

[47]:
```
from sklearn.svm import SVC

poly_svm = SVC(kernel='poly')

poly_svm.fit(X_train, y_train)
```

[47]:
```
SVC(kernel='poly')
```

[48]:
```
polysvm_pred = poly_svm.predict(X_test)

accuracy_score(y_test, polysvm_pred)
```

[48]:
```
0.9736842105263158
```

[49]:
```
rbf_svm = SVC(kernel='rbf')

rbf_svm.fit(X_train, y_train)
```

[49]:
```
SVC()
```

[50]:
```
rbfsvm_pred = rbf_svm.predict(X_test)

accuracy_score(y_test, rbfsvm_pred)
```

[50]:
```
0.9736842105263158
```

[51]:
```
confusion_matrix(y_test, rbfsvm_pred)
```

[51]:
```
array([[12,  0,  0],
       [ 0, 12,  1],
       [ 0,  0, 13]], dtype=int64)
```

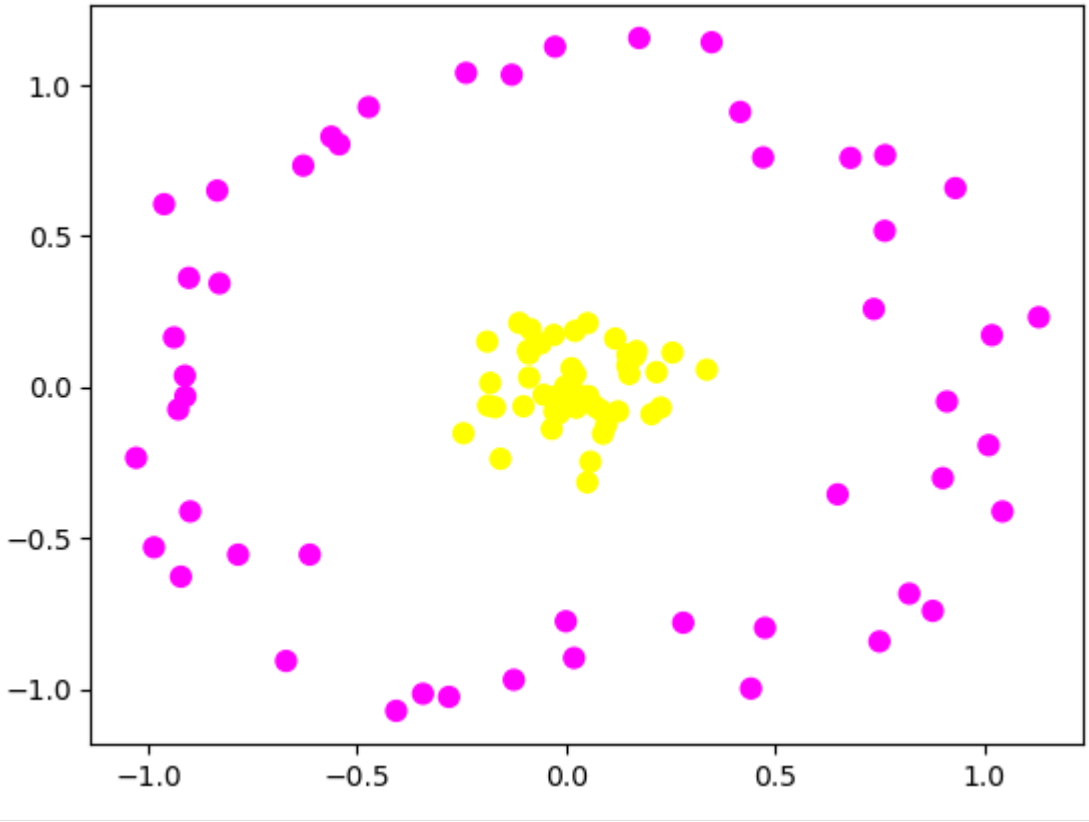
Therefore, for the Iris dataset the linear kernel has the best performance, indicating that the data is linearly separable.

Due to supporting different types of kernels, SVM can tackle different kinds of datasets, both linear and non-linear. In the real world, many datasets are not linear. If we cannot get good results with linear models, it helps to apply SVM with non-linear kernels.

Let's compare two SVM models using a linear and rbf kernel to the following data which is obviously non-linearly separable.

[52]:
```
from sklearn.datasets import make_circles
X2, y2 = make_circles(100, factor=.1, noise=.1)
plt.scatter(X2[:, 0], X2[:, 1], c=y2, s=50, cmap='spring')
```

[52]:
```
<matplotlib.collections.PathCollection at 0x21948f4ea90>
```



[53]:
```
X2_train, X2_test, y2_train, y2_test = train_test_split(X2, y2, random_state=123, test_
↪size=0.25)
```

[54]:
```
rbf_svm = SVC(kernel='rbf')

rbf_svm.fit(X2_train, y2_train)
```

[54]:
```
SVC()
```

[55]:
```
rbfsvm_pred = rbf_svm.predict(X2_test)

accuracy_score(y2_test, rbfsvm_pred)
```

[55]:
```
1.0
```

[56]:
```
lin_svm_2 = SVC(kernel='linear')
lin_svm_2 .fit(X2_train, y2_train)
```

[56]:
```
SVC(kernel='linear')
```

[57]:
```
linsvm_pred = lin_svm_2.predict(X2_test)

accuracy_score(y2_test, linsvm_pred)
```

[57]:
```
0.6
```

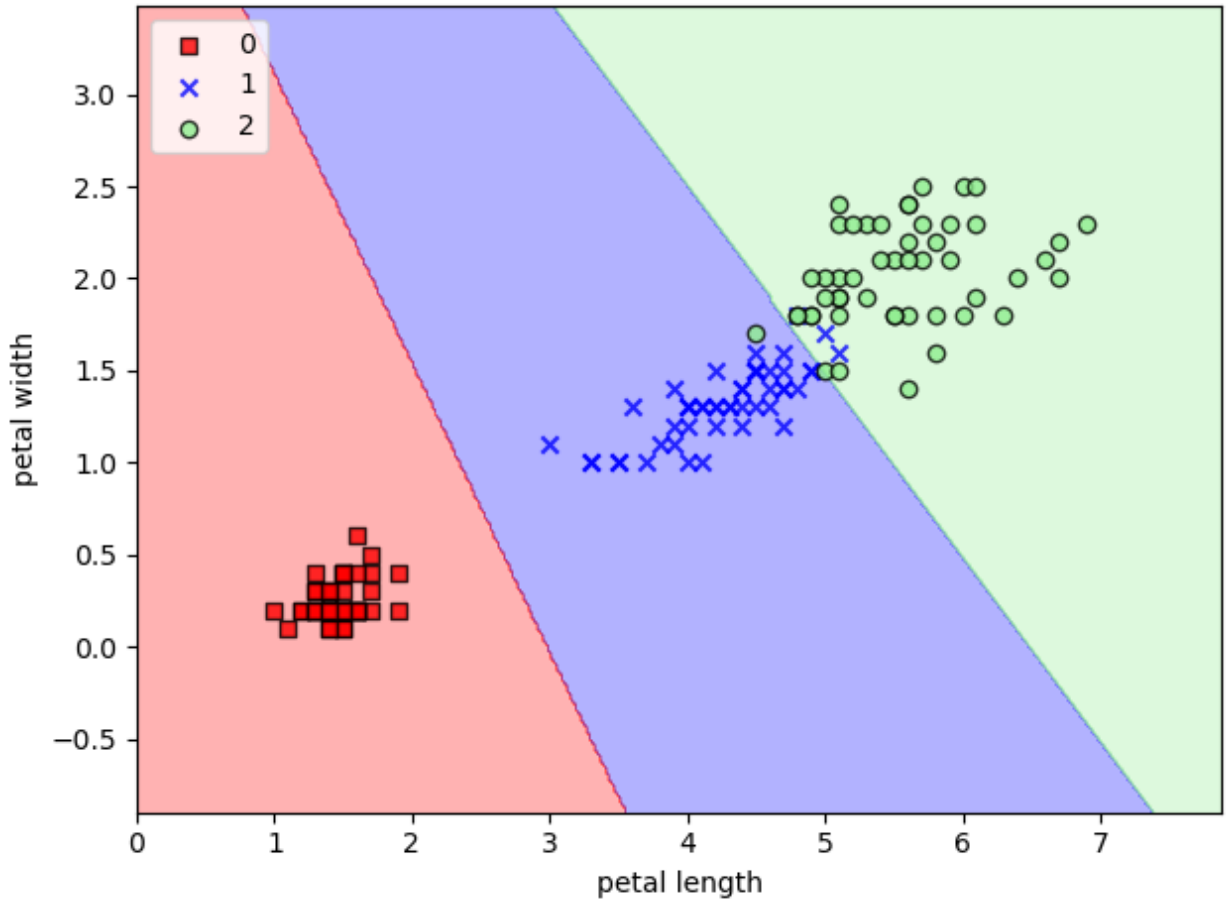
As expected, the model with the rbf kernel performed much better than the linear model.

Plot the Decision Boundary

We can also plot the decision boundary again when using two features of the dataset. Note that in this case because we use only two features, the accuracy of linear SVM is not 100%.

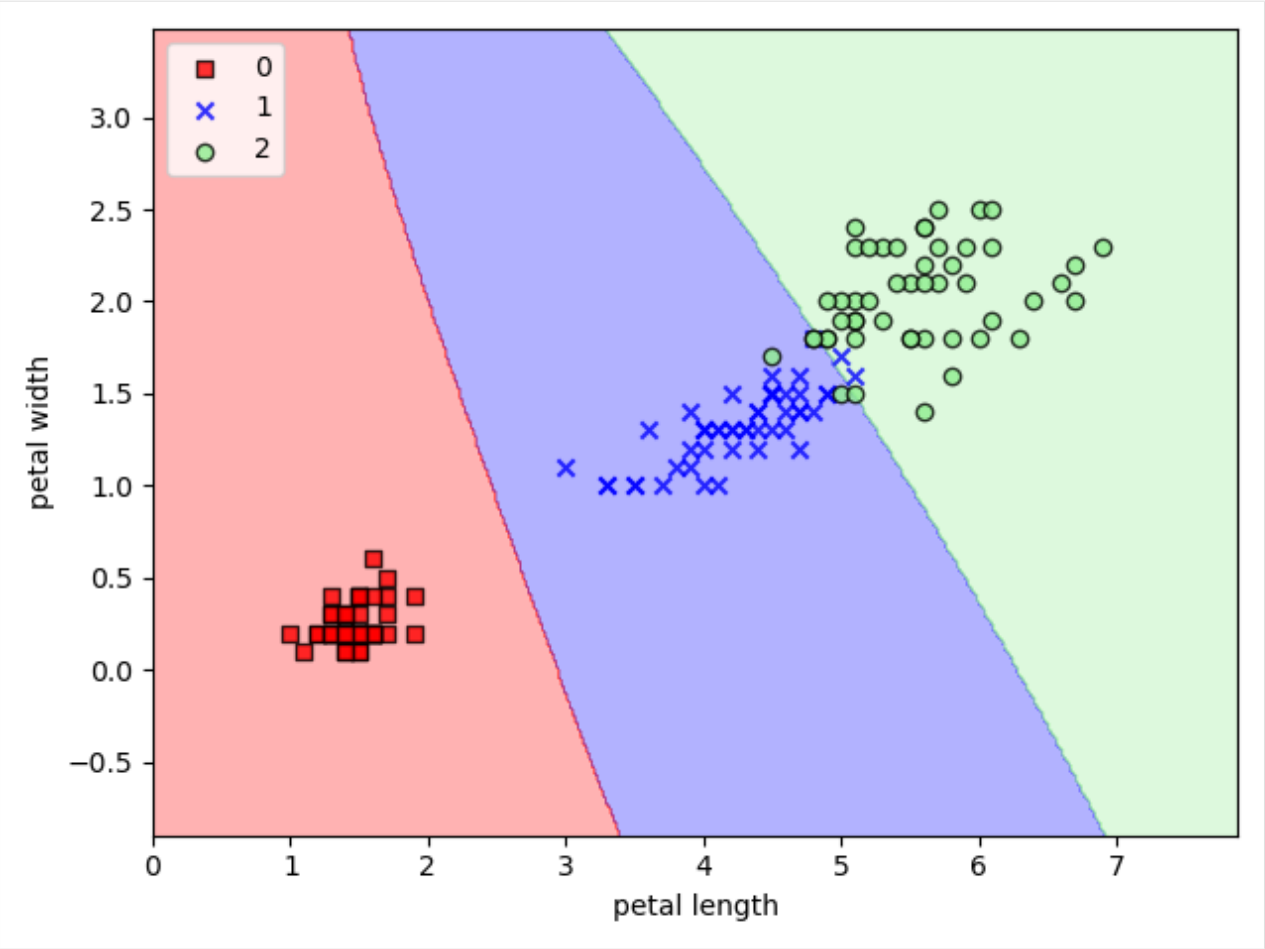
[58]:
```
lin_svm_3 = SVC(kernel='linear')

lin_svm_3.fit(X1_train, y1_train)

accuracy = lin_svm_3.score(X1_test, y1_test)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))

plot_decision_regions(X1, y1, classifier=lin_svm_3)
```

```
The test accuracy is 97.37 %
```



[59]:
```
rbf_svm_3 = SVC(kernel='rbf')

rbf_svm_3.fit(X1_train, y1_train)

accuracy = rbf_svm_3.score(X1_test, y1_test)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))

plot_decision_regions(X1, y1, classifier=rbf_svm_3)
```

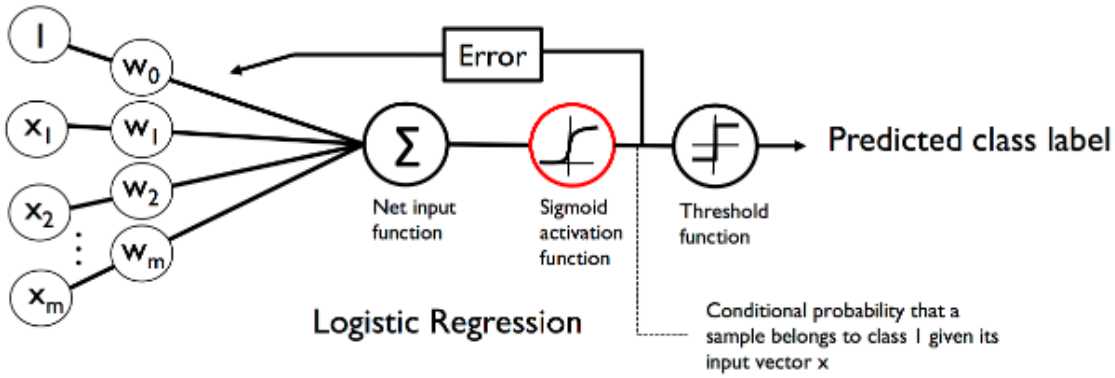
```
The test accuracy is 94.74 %
```



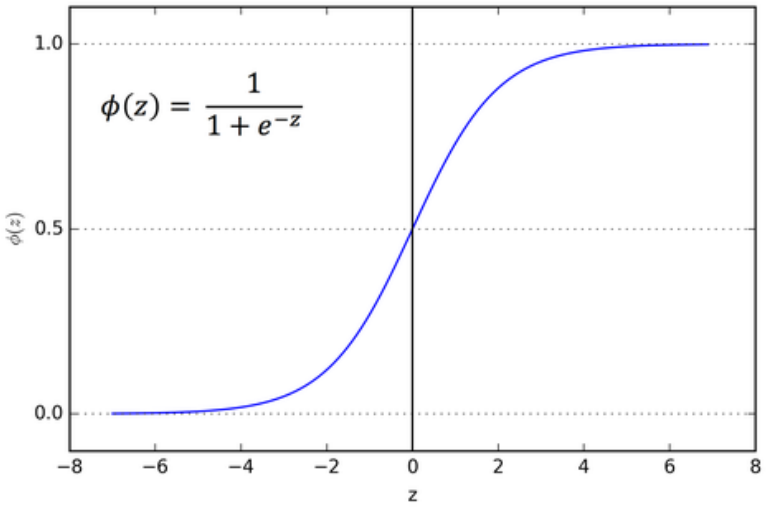
13.2.3 Logistic Regression

Logistic Regression is one of the most frequently used classifiers, because it is simple and fast, it performs well on linearly separable data, and it can produce good baseline performance. The name can be confusing, because Logistic Regression is a classification algorithm, and not a regression algorithm.

The main concept of classification with Logistic Regression is shown in the next figure. Namely, for a set of input features $x_0, x_1, ..., x_m$, the algorithm calculates a set of weights $w_0, w_1, ..., w_m$. The product of the input features and weights $\boldsymbol{w^T} * \boldsymbol{x}$ is passed through a **logistic sigmoid activation function**, also simply known as sigmoid function. The parameters (weights) of Logistic Regression $w_0, w_1, ..., w_m$ are learned by applying an optimization algorithm (e.g., gradient descent, or Newton methods), and it can be considered a simple type of neural network with one layer.



The plot of a sigmoid function is shown on the next graph, for an input $z = \boldsymbol{w}^T * \boldsymbol{x}$. The function squeezes all inputs into the [0,1] range. The output of the sigmoid function can be interpreted as a probability that the data point belongs to class 1 or 0 in binary classification problems (two classes).



For multiclass classification problems, the *softmax function* is used as a generalized version of the sigmoid function, and it outputs the probability that a data point belongs to multiple classes.

Let's fit a Logistic Regression model to the Iris dataset.

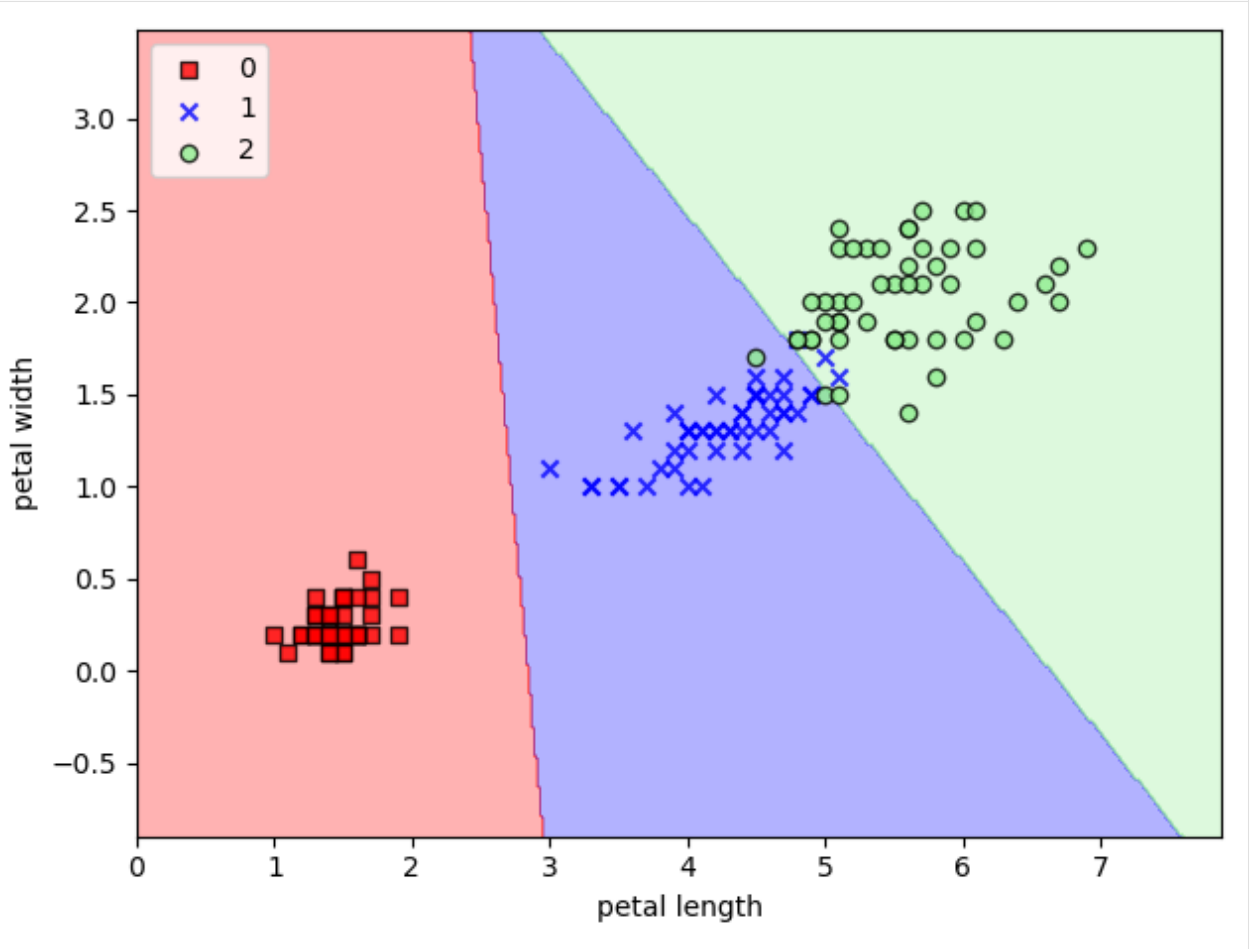
[60]:
```
from sklearn.linear_model import LogisticRegression

lr_model_1 = LogisticRegression()
lr_model_1.fit(X_train, y_train)

accuracy = lr_model_1.score(X_test, y_test)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))
```

```
The test accuracy is 97.37 %
```

Plot the Decision Boundary

[61]:
```
lr_model_2 = LogisticRegression()

lr_model_2.fit(X1_train, y1_train)

accuracy = lr_model_2.score(X1_test, y1_test)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))

plot_decision_regions(X1, y1, classifier=lr_model_2)
```

```
The test accuracy is 97.37 %
```



13.2.4 Decision Trees

Decision trees is a machine learning algorithm based on applying an intuitive way to classify or label objects, by simply asking a series of questions designed to help the classification. A simple example is shown below, where the decision tree starts from the root node and splits the data at each next node based on selected features and a threshold value. The nodes create the branches of the tree.

The total number of times the nodes are split is the *depth* of the tree. We can typically set a limit for the maximum depth of the tree. The ending nodes in the tree are called leaf nodes.

Decision trees can be applied for classification and regression of numerical and categorical features. For numerical

data, the same concept applies, where the tree can split the data based on selected features and threshold values.

To make predictions on new data, we follow the branches from the root node to a leaf node by following the splits based on the values of the input features. The predicted class is the value associated with the leaf node reached.

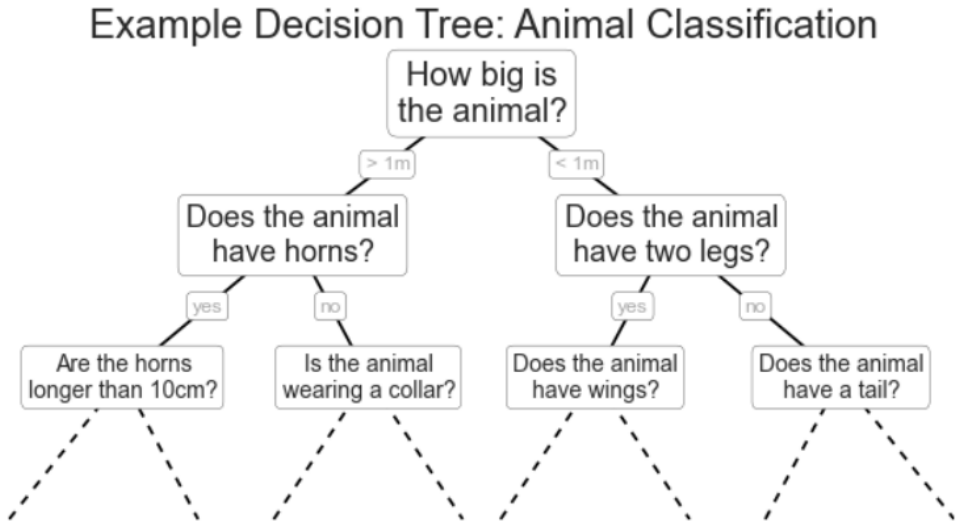


Figure source: Reference [1].

The trick in splitting the nodes is to ask the right questions. In training a decision tree classifier, the algorithm looks at the data and selects the features and threshold values that contain the most information (i.e., maximize the information gain). Several splitting criteria are available in scikit-learn. In the following example, `criterion='gini'` indicates that the "gini impurity" criterion is used.

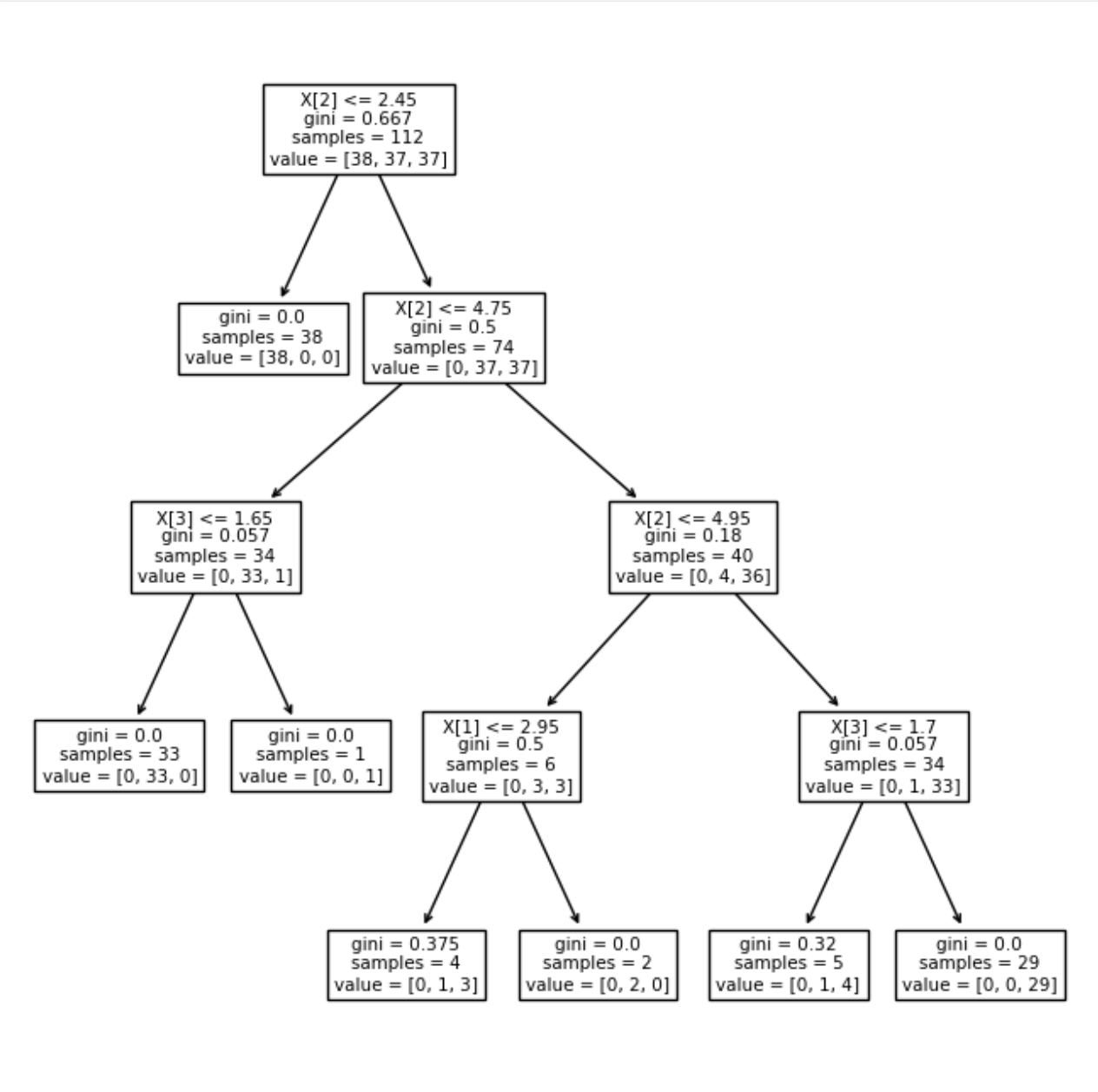
Here is an example of training a decision tree classifier in scikit-learn on the Iris dataset.

[62]:
```
from sklearn.tree import DecisionTreeClassifier

tree_model_1 = DecisionTreeClassifier(criterion='gini', max_depth=4)

tree_model_1.fit(X_train, y_train)

tree_pred = tree_model_1.predict(X_test)

accuracy = accuracy_score(y_test, tree_pred)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))
```

```
The test accuracy is 89.47 %
```

Scikit-learn has a function for plotting decision trees. In the plot below, we can see that splitting was done mostly based on the features `X[2]` and `X[3]`. For instance, in the root node if the petal length is < 2.45 cm, then split the node, etc.

[63]:
```
from sklearn import tree

plt.figure(figsize=(8,8))
tree.plot_tree(tree_model_1)
plt.show()
```



Plot the Decision Boundary

The decision boundary is shown in the figure below. The decision tree algorithm separates the data by dividing the space of the input features into rectangles.

One issue with Decision Trees is that it is easy to create trees that overfit the data. That is, such models can fit the training data almost perfectly, to the point where they begin fitting the noise in the data. This can be seen below for the decision boundary for the second class.

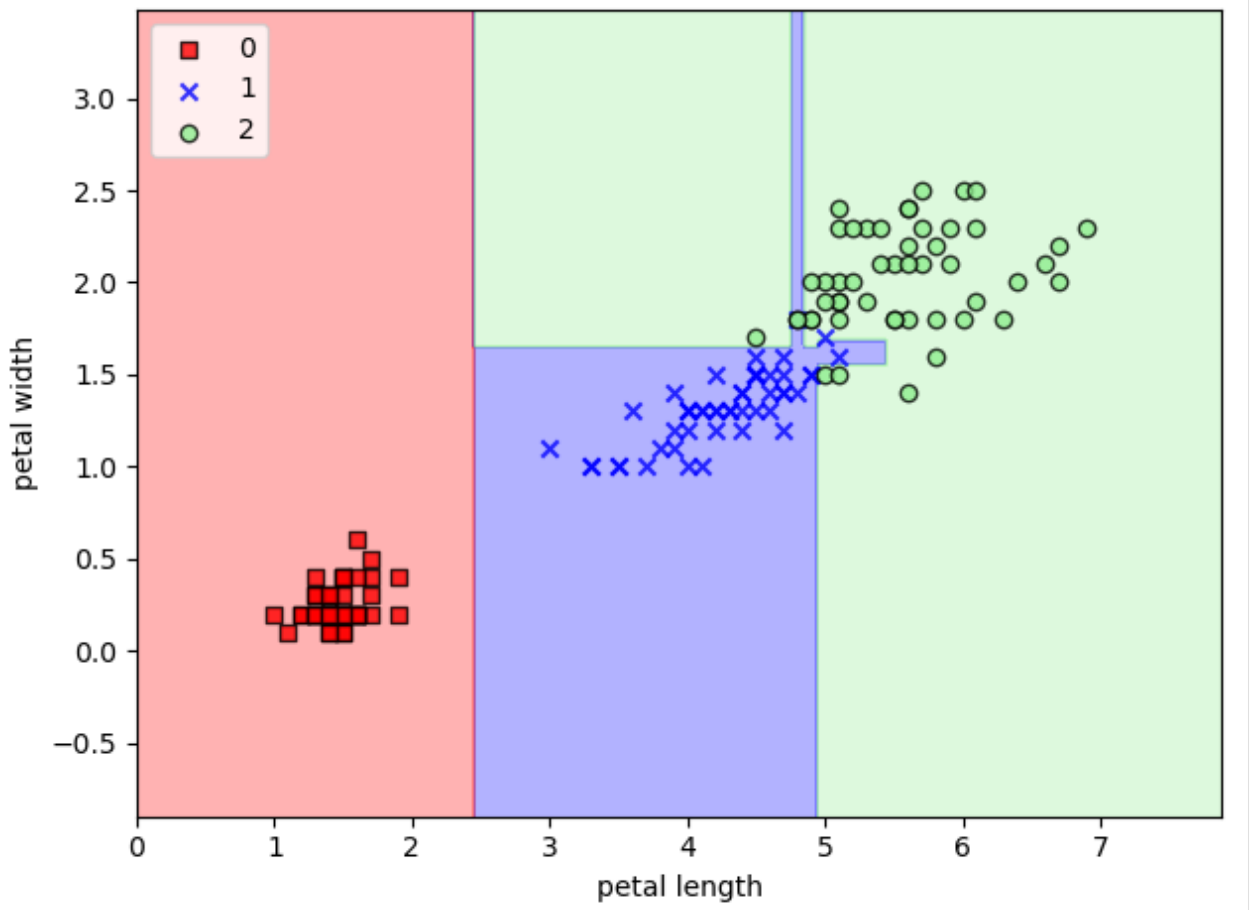
[64]:
```
tree_model_2 = DecisionTreeClassifier(criterion='gini', max_depth=6)

tree_model_2.fit(X1_train, y1_train)
```

(continues on next page)

(continued from previous page)

```
tree_pred = tree_model_2.predict(X1_test)

accuracy = accuracy_score(y1_test, tree_pred)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))

plot_decision_regions(X1, y1, classifier=tree_model_2)
```

```
The test accuracy is 94.74 %
```



One possible way to address overfitting is to use an **ensemble method**, which essentially averages the results of many individual estimators that overfit the data. The resulting ensemble estimators are usually more robust and accurate than the individual estimators which make them up.

One of the most common ensemble methods is Random Forest, in which the ensemble is made up of many decision trees. This is the topic of the next section.

13.2.5 Random Forest

As we mentioned, **Random Forest** uses many decision trees (`n_estimators` below) to create a more robust model that has reduced overfitting on the data. Typically, using larger number of estimators results in improved performance, but increased computational cost.

In the next cell, we used 100 estimators (decision trees). The parameter `n_jobs` specify the number of CPU cores used for speeding up the model training by using parallel processing, where different cores can process different decision trees.

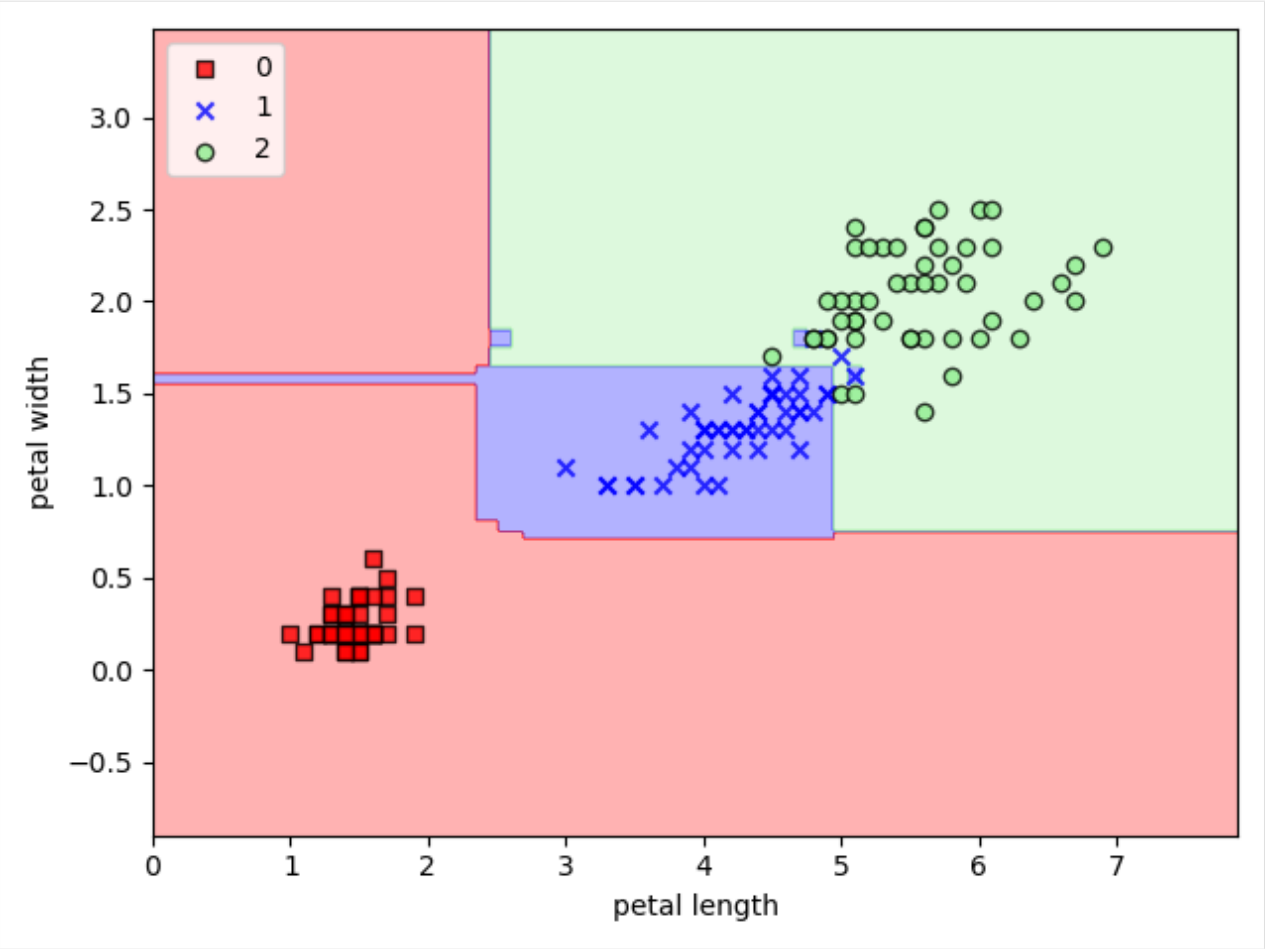
[65]:
```
from sklearn.ensemble import RandomForestClassifier

rf_model_1 = RandomForestClassifier(criterion='gini', n_estimators=100, random_state=1,␣
↪n_jobs=2)

rf_model_1.fit(X_train, y_train)

rf_pred = rf_model_1.predict(X_test)

accuracy = accuracy_score(y_test, rf_pred)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))
```

```
The test accuracy is 97.37 %
```

Plot the Decision Boundary

[66]:
```
rf_model_2 = RandomForestClassifier(criterion='gini', n_estimators=100, random_state=1,␣
↪n_jobs=2)

rf_model_2.fit(X1_train, y1_train)

rf_pred = rf_model_2.predict(X1_test)

accuracy = accuracy_score(y1_test, rf_pred)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))

plot_decision_regions(X1, y1, classifier=rf_model_2)
```

```
The test accuracy is 97.37 %
```



13.2.6 Naive Bayes

Naive Bayes are a set of learning algorithms that are based on applying Bayes' theorem. According to the theorem, the posterior probability of the class label y given input features $x_1, \ldots, x_n$ can be calculated based on the prior probability of the class $P(y)$, the likelihood of the training data given the class label $P(x_1, \ldots, x_n \mid y)$, and the evidence $P(x_1, \ldots, x_n)$.

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots, x_n \mid y)}{P(x_1, \ldots, x_n)}$$

This method is called Naive Bayes, because it is based on a "naive" assumption that the input features are independent given the class label.

The implementation is shown below.

[67]:
```
from sklearn.naive_bayes import GaussianNB

nb_model_1 = GaussianNB()

nb_model_1.fit(X_train, y_train)

nb_pred = nb_model_1.predict(X_test)
```

(continues on next page)

(continued from previous page)

```
accuracy = accuracy_score(y_test, nb_pred)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))
```

```
The test accuracy is 97.37 %
```

Scikit-learn offers several different implementations of Naive Bayes, which differ based on the assumption made about the distribution of the training data. These include: GaussianNB (typically used with continuous features that have normal distribution), BernoulliNB (used with binary features), CategoricalNB (used with categorical features), and MutlinomialNB (used with discrete features that represent counts or frequencies).

13.2.7 Perceptron

Perceptron classification algorithm has similarities to the Logistic Regression classifier, and they are both linear classifiers that can be seen as a simple one-layer neural network. Both algorithms learn a set of weights and use similar learning strategies. One difference is that Perceptron uses a step threshold function to output the class prediction, and Logistic Regression uses a sigmoid logistic function to output the class prediction.

Perceptron outputs a binary class label based on whether the product of input and weights is above or below a certain threshold. The parameters in Perceptron are updated via a simple learning algorithm. Whenever a misclassification occurs, the algorithm tries to minimize classification errors.

Perceptron is most suitable for binary classification of linearly separable data.

The following figure depicts classification with Perceptron.

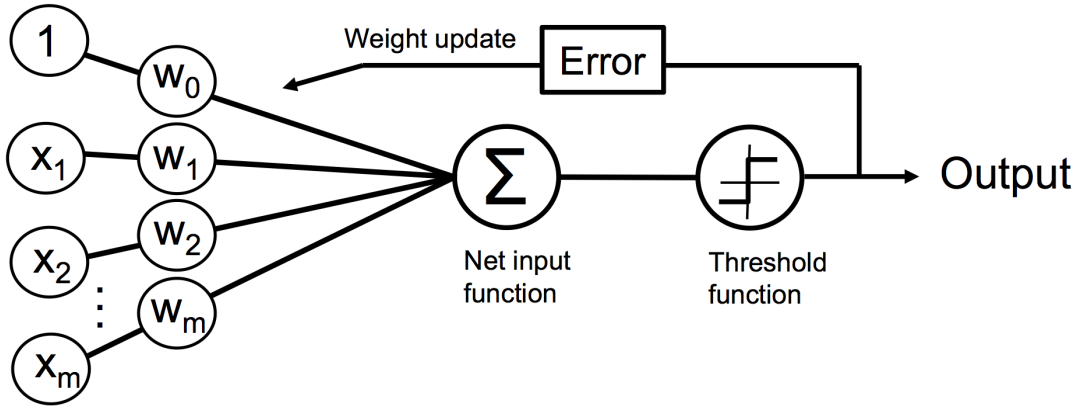


Figure source: Reference [2].

In the model below, the argument `max_iter` is the number of epochs (i.e., the number of passes through the training data), and `eta0` is the learning rate of the model.

[68]:
```
from sklearn.linear_model import Perceptron

ppn_model_1 = Perceptron(max_iter=40, eta0=0.1)

ppn_model_1.fit(X_train, y_train)

ppn_pred = ppn_model_1.predict(X_test)

accuracy = accuracy_score(y_test, ppn_pred)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))
```

```
The test accuracy is 65.79 %
```

This type of learning algorithms require more finetuning of the hyperparameters. One of the most important parameters is the learning rate (`eta0` in this case). Let's try to change it.

[69]:
```
ppn_model_2 = Perceptron(max_iter=40, eta0=0.001)

ppn_model_2.fit(X_train, y_train)

ppn_pred = ppn_model_2.predict(X_test)

accuracy = accuracy_score(y_test, ppn_pred)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))
```

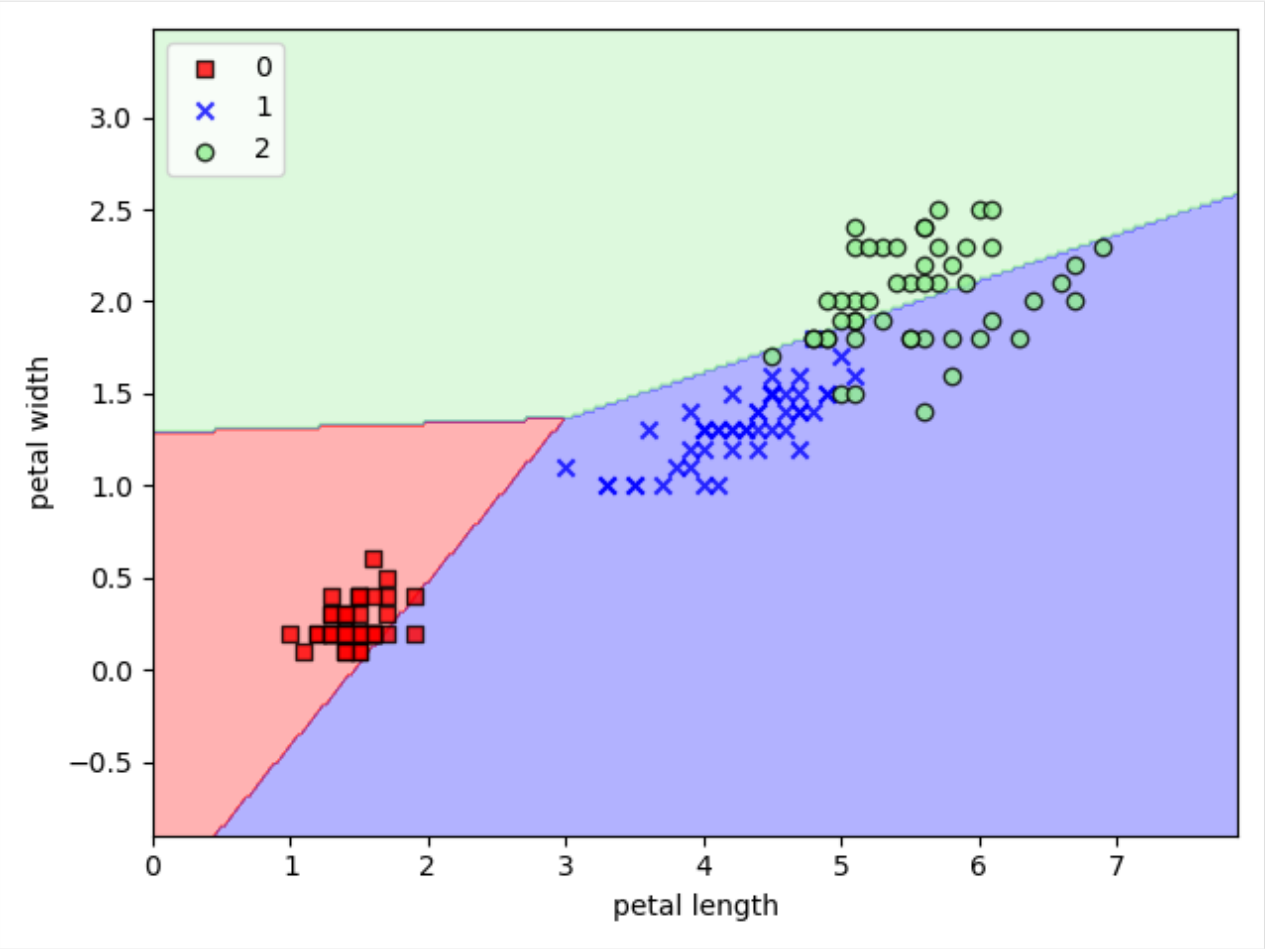
```
The test accuracy is 78.95 %
```

Although the accuracy improved with different learning rate, the performance is still not very good. One reason may be because this algorithm is more sensitive to the range of input data, and we may need to scale the training data to obtain better performance.

Plot the Decision Boundary

[70]:
```
ppn_model_3 = Perceptron(max_iter=40, eta0=0.001)

ppn_model_3.fit(X1_train, y1_train)

ppn_pred = ppn_model_3.predict(X1_test)

accuracy = accuracy_score(y1_test, ppn_pred)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))

plot_decision_regions(X1, y1, classifier=ppn_model_3)
```

```
The test accuracy is 81.58 %
```



13.2.8 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is another popular estimator in scikit-learn. In fact, SGD is an optimization technique, and it does not correspond to a specific family of machine learning models. In other words, SGD is just an approach to train a model using stochastic gradient descent to learn an optimal set of model parameters that minimize a loss function.

SGD in scikit-learn supports several loss functions, and depending on the loss parameters that we select, it is used for different types of classification tasks. For instance, if we use `loss='perceptron'` it will be equivalent to solving the Perceptron algorithm with SGD optimization. Using `loss='log'` is equivalent to training Logistic Regression with SGD (note that by default Logistic Regression in scikit-learn uses a different optimization algorithm, called BFGS algorithm). Similarly, using `loss='hinge'` corresponds to linear SVM.

[71]:
```
from sklearn.linear_model import SGDClassifier

sgd_model_1 = SGDClassifier(max_iter=80, loss='hinge', random_state=1)

sgd_model_1 .fit(X_train, y_train)

sgd_pred = sgd_model_1 .predict(X_test)

accuracy = accuracy_score(y_test, sgd_pred)
```

(continues on next page)

(continued from previous page)

```
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))
```

```
The test accuracy is 92.11 %
```

[72]:
```
from sklearn.linear_model import SGDClassifier

sgd_model_2 = SGDClassifier(max_iter=40, loss='perceptron', random_state=1)

sgd_model_2.fit(X_train, y_train)

sgd_pred = sgd_model_2.predict(X_test)

accuracy = accuracy_score(y_test, sgd_pred)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))
```

```
The test accuracy is 65.79 %
```

Similar to Perceptron, SGD is sensitive to feature scaling. And also, scikit-learn offers SGDRegressor for regression tasks.

7.13.3 13.3 Supervised Learning: Regression

Linear Regression

Estimator functions in scikit-learn are implemented in a very simmilar way as the classification estimators.

For instance, a Linear Regression estimator is imported and implemented in scikit-learn as follows.

[73]:
```
from sklearn.linear_model import LinearRegression
```

In the following cell, a new instance of a Linear Regression estimator is created, and we named it `lr_model`.

[74]:
```
lr_model = LinearRegression()
```

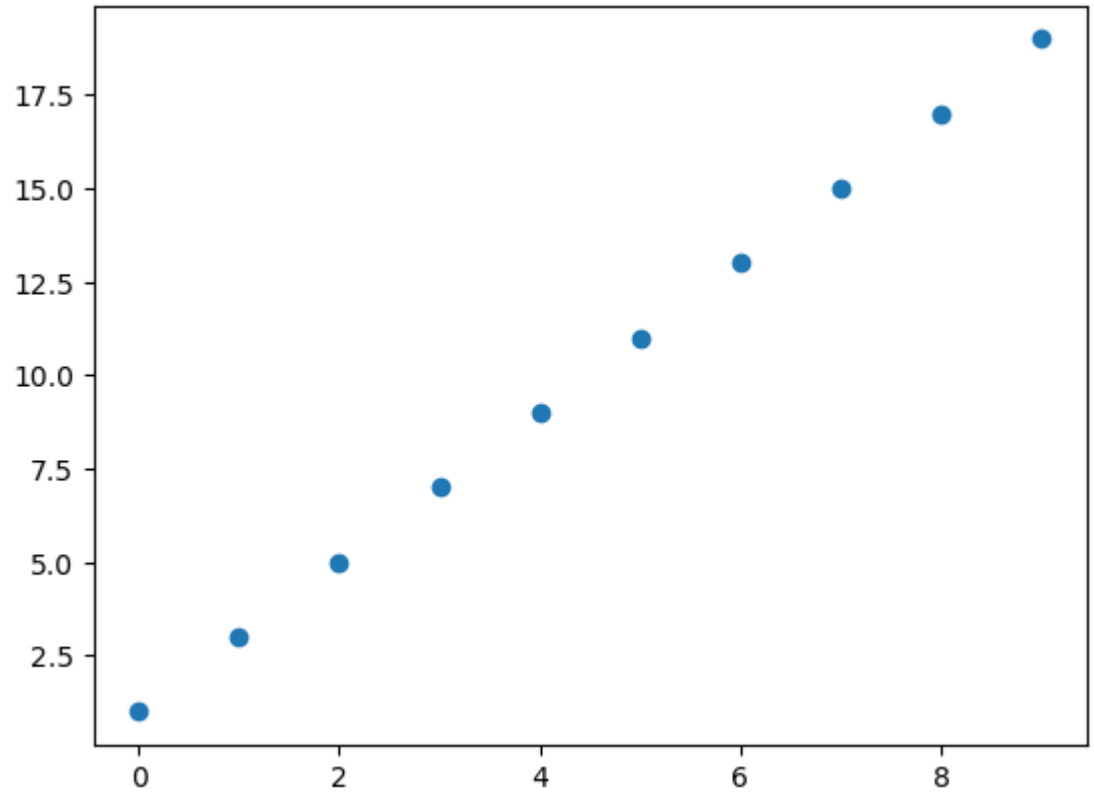
[75]:
```
print(lr_model)
```

```
LinearRegression()
```

Let's create data for fitting the Linear Regression model. For example, we can create data for a line `2 * x + 1`.

[76]:
```
x = np.arange(10)
y = 2 * x + 1

print(x)
print(y)

plt.plot(x, y, 'o')
plt.show()
```

```
[0 1 2 3 4 5 6 7 8 9]
[ 1  3  5  7  9 11 13 15 17 19]
```



Since the input data in scikit-learn needs to be a 2-dimensional array, we will add a second dimension to `X` with `np.newaxis`.

[77]:
```
# The input data needs to be a 2D array
X = x[:, np.newaxis]
print(X.shape)
print(y.shape)
```

```
(10, 1)
(10,)
```

The next line fits the model to the data.

[78]:
```
# fit the model on our data
lr_model.fit(X, y)
```

[78]:
```
LinearRegression()
```

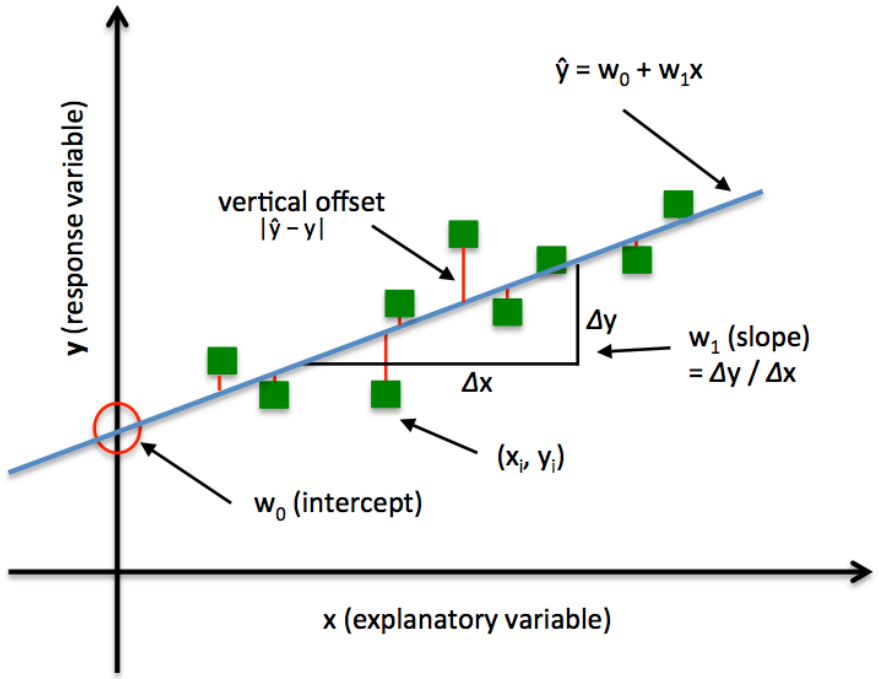
The learned model parameters are shown next.

[79]:
```
# underscore at the end indicates a fit parameter
print(lr_model.coef_)
print(lr_model.intercept_)
```

```
[2.]
1.0000000000000053
```

In linear regression, the goal is to learn a linear model (line) that fits the input data in an optimal way. The estimator has two parameters: coefficient (the slope of the line) and intercept (the point of interception of the y-axis). The algorithm

solves the linear regression by minimizing the deviation of the input data from the line.

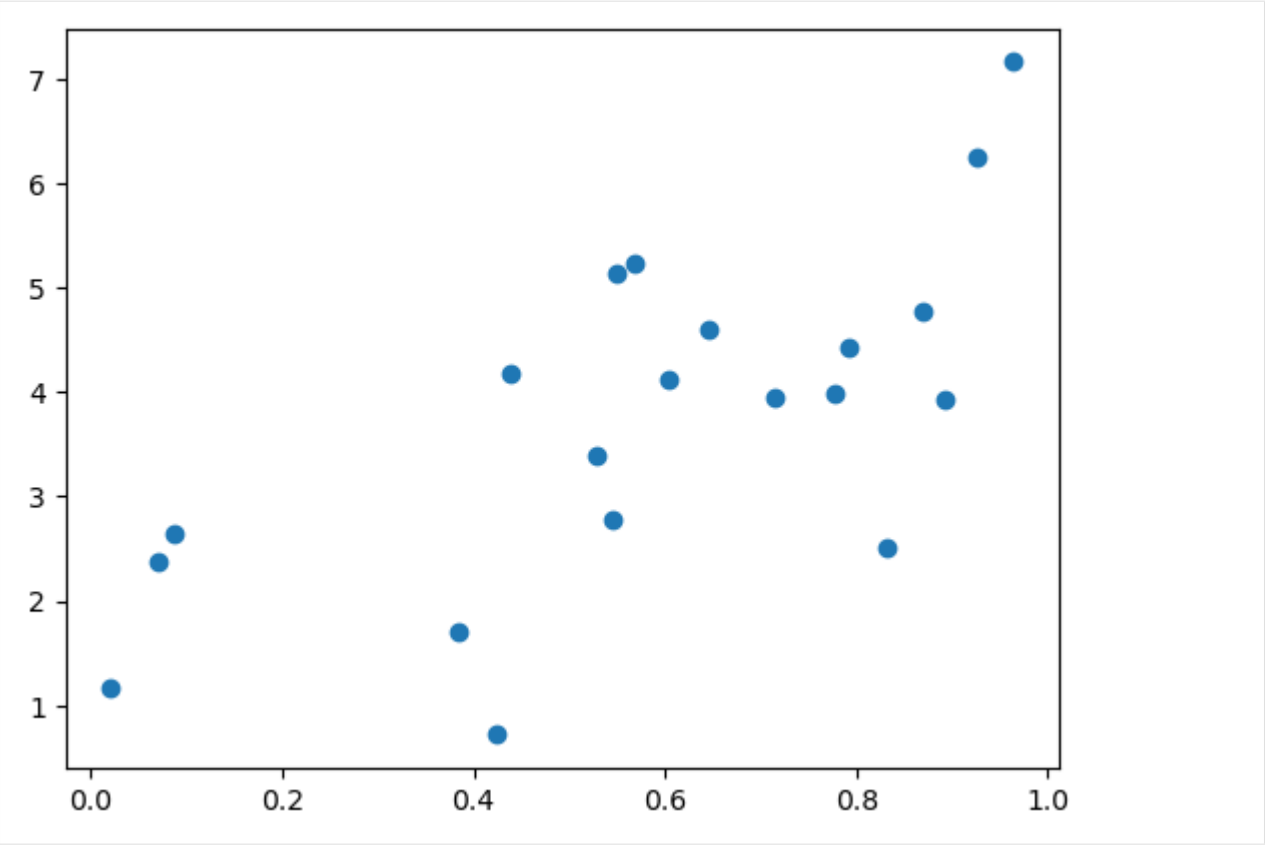


The two parameters shown in the above cell are correctly estimated, since the data was generated from the line `2 * x + 1`, and the model found a line with a slope 2 and intercept of approximately 1, as expected.

Linear Regression - Example 2

Let's look at another example where the data does not come from a straight line.

[80]:
```
# Create data
np.random.seed(0)
X = np.random.random(size=(20, 1))
y = 3 * X.squeeze() + 2 + np.random.randn(20)

plt.plot(X.squeeze(), y, 'o')
plt.show()
```



Let's apply Linear Regression to obtain a new model that fits the data.

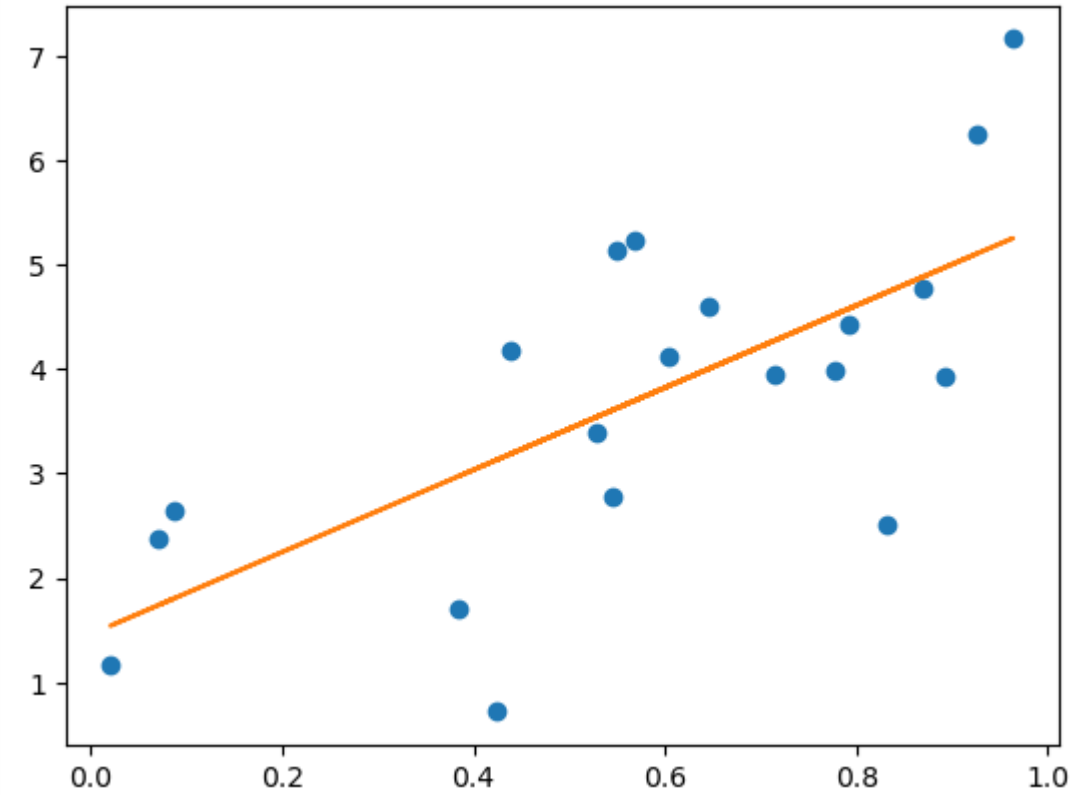
[81]:
```
lr_model_2 = LinearRegression()
lr_model_2.fit(X, y)
```

[81]:
```
LinearRegression()
```

We can predict the target values `y_predicted` for given input values `X`, as with classifier estimators.

[82]:
```
y_predicted = lr_model_2.predict(X)
```

[83]:
```
plt.plot(X, y, 'o') # plot the training data with circle markers
plt.plot(X, y_predicted)     # plot the predicted line
plt.show()
```

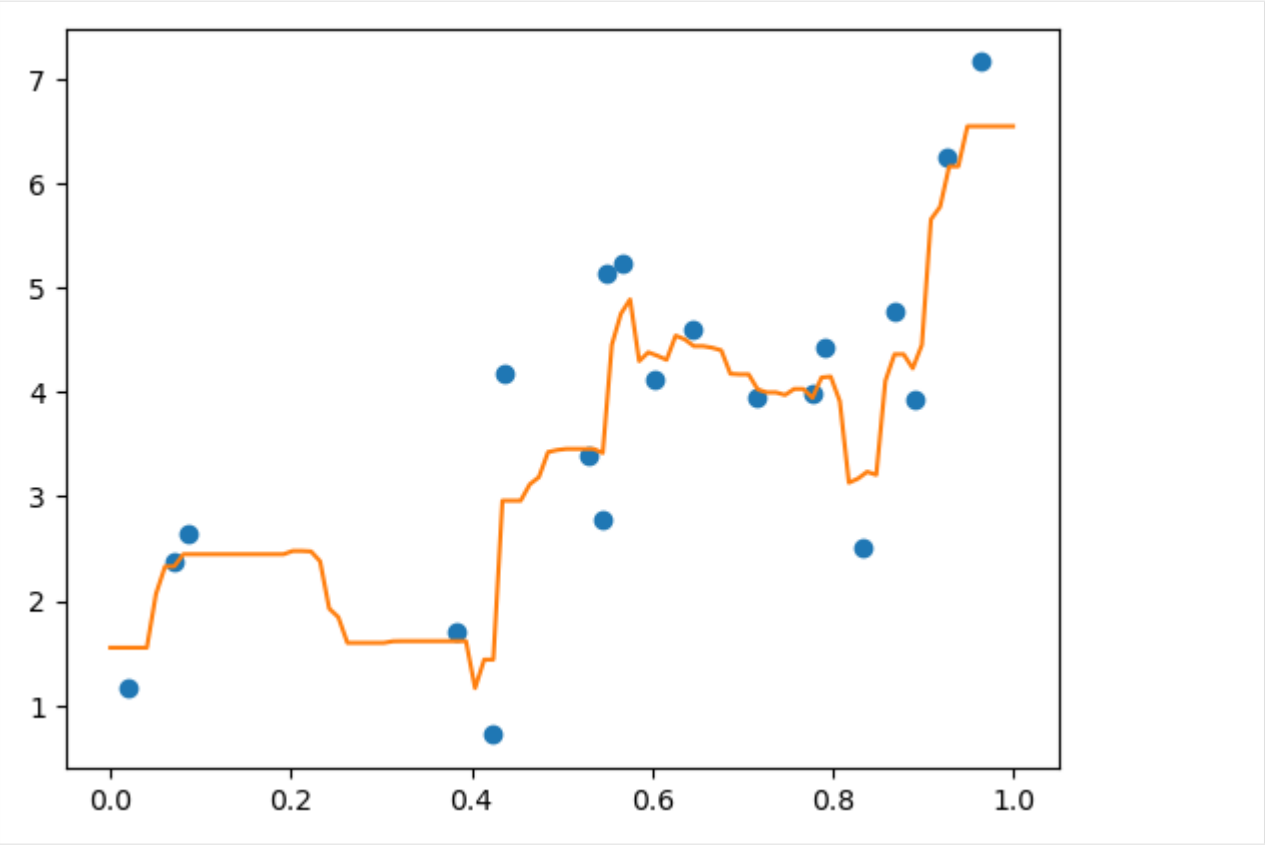


Regression with Random Forest

Besides Linear Regression, scikit-learn also offers many more sophisticated models for regression. In the next example, a Random Forest regression model is used to fit the data that we created. The output of the model is non-linear, and hence, this model fits better the data than the Linear Regression model.

[84]:
```
# Fit a Random Forest
from sklearn.ensemble import RandomForestRegressor
rf_model = RandomForestRegressor()
rf_model.fit(X, y)

X_fit = np.linspace(0, 1, 100)[:, np.newaxis]
y_predicted = rf_model.predict(X_fit)

# Plot the data and the model prediction
plt.plot(X, y, 'o')
plt.plot(X_fit.squeeze(), y_predicted)
```

[84]:
```
[<matplotlib.lines.Line2D at 0x2194acb6880>]
```



7.13.4 13.4 Unsupervised Learning: Clustering

Unsupervised learning algorithms employ data without labels, and often focus on finding similarities or patterns in the set of samples. Unsupervised learning comprises tasks such as *dimensionality reduction*, *clustering*, and *density estimation*.

There are also *semi-supervised learning* approaches, where for example, unsupervised learning can be used to find informative features in data, and then these features can be used within a supervised learning framework.

Clustering with k-Means

Clustering methods find *clusters* in the data, i.e., they group together data points that are homogeneous with respect to a given criterion.

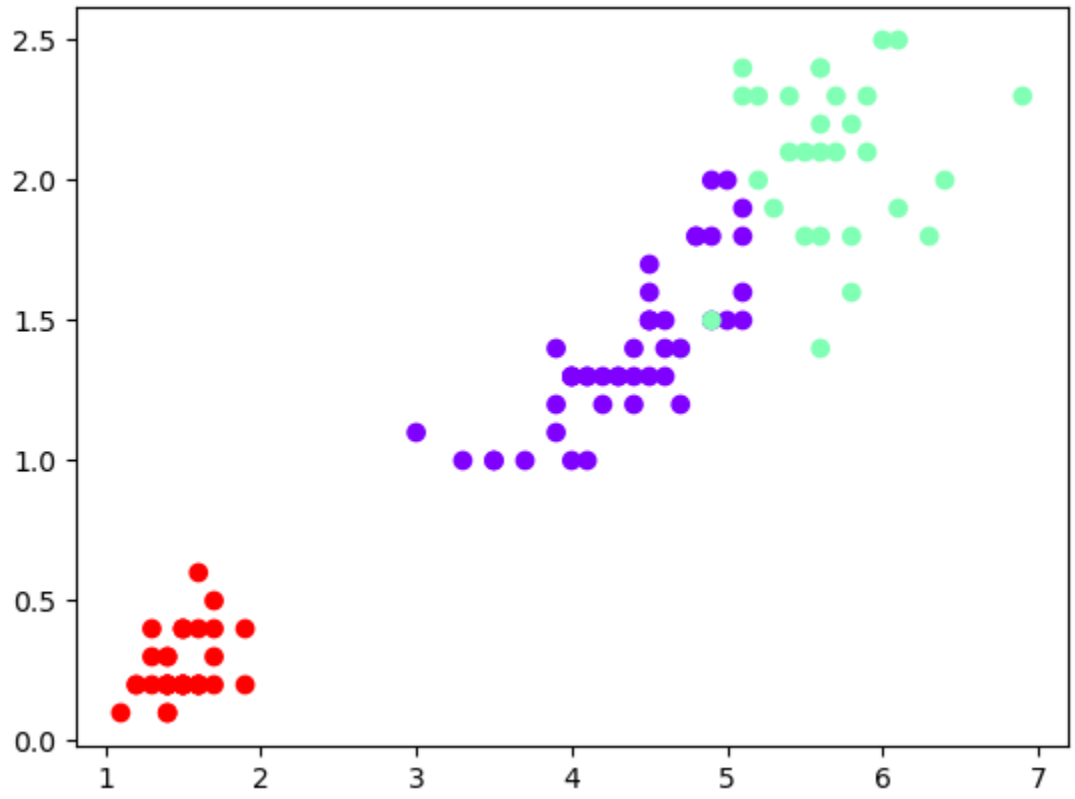
A popular clustering algorithm is *k*-means clustering, which iteratively calculates distances from data points to centroids of clusters, until the distances from all data points to the centroids of the clusters are minimized. It employs the Expectation Maximization approach to cluster the data points.

[85]:
```
from sklearn.cluster import KMeans

k_means = KMeans(n_clusters=3, random_state=0)

k_means.fit(X_train)

y_pred = k_means.predict(X_train)
```

(continues on next page)

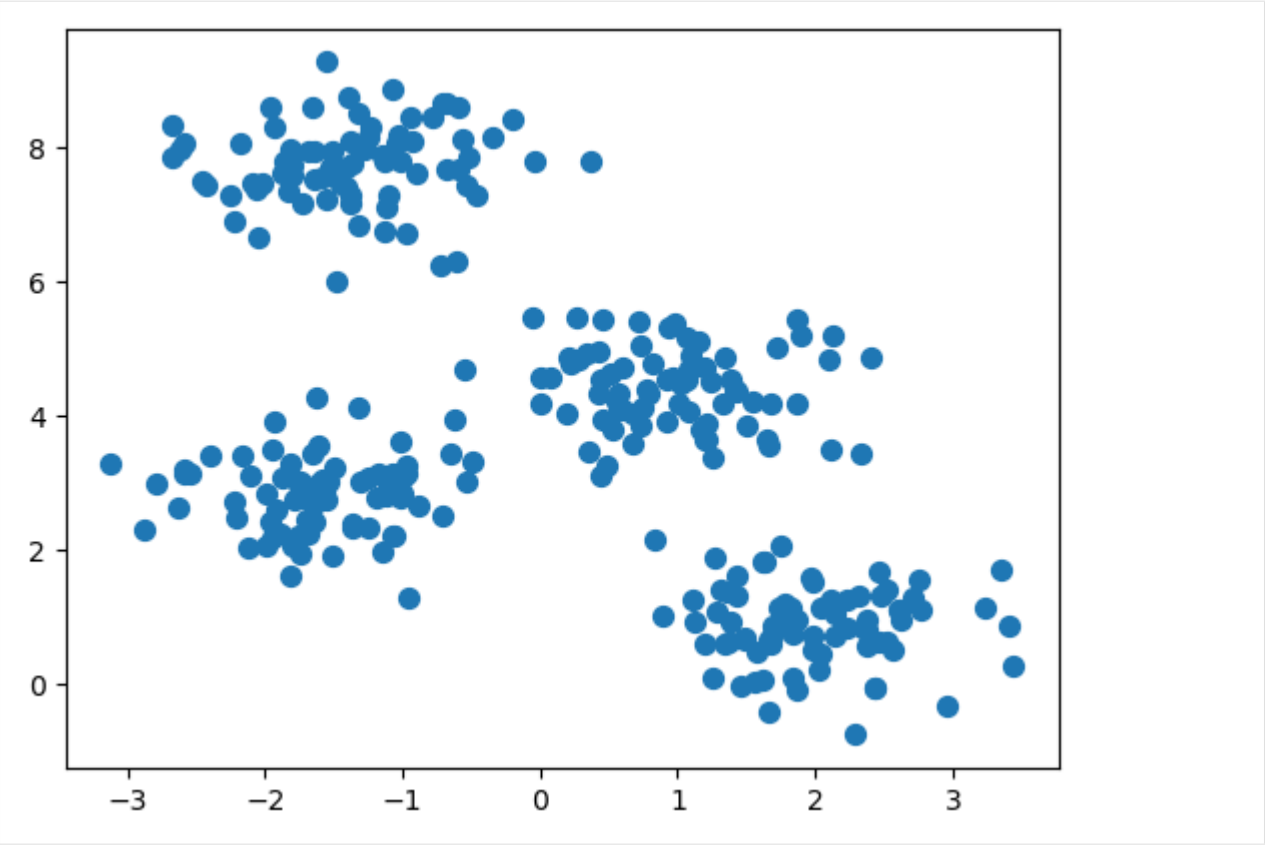
(continued from previous page)

```
plt.scatter(X_train[:, 2], X_train[:, 3], c=y_pred, cmap='rainbow');
```

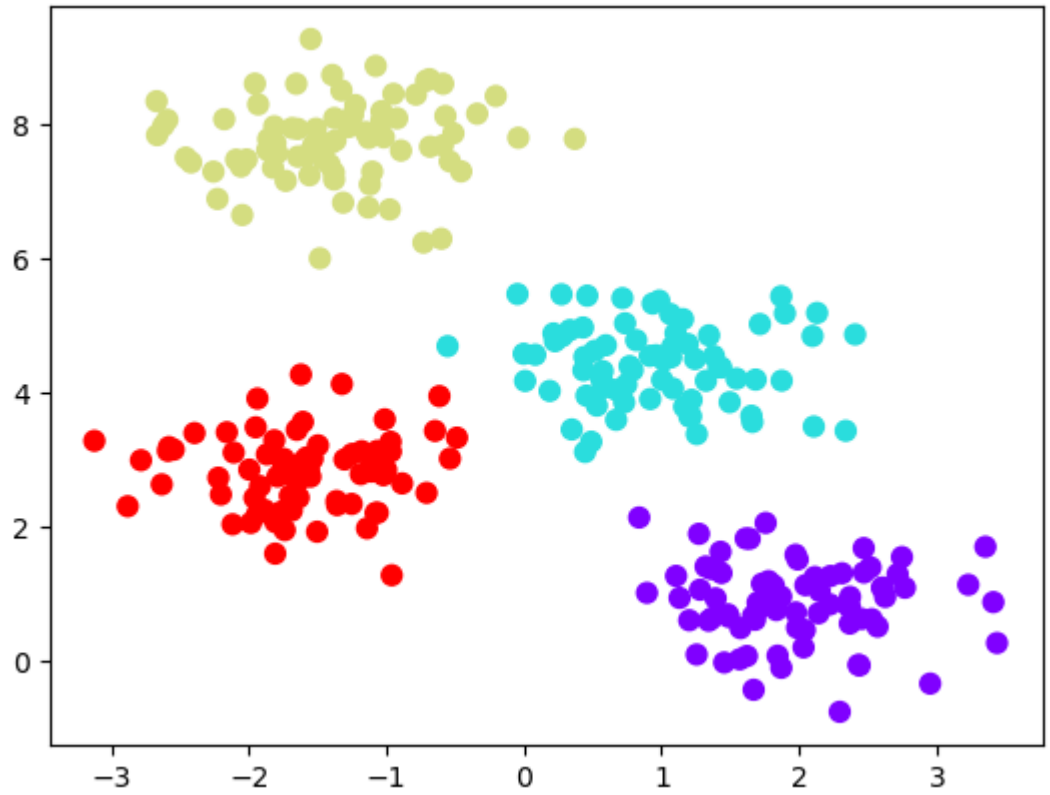


Here is one more example.

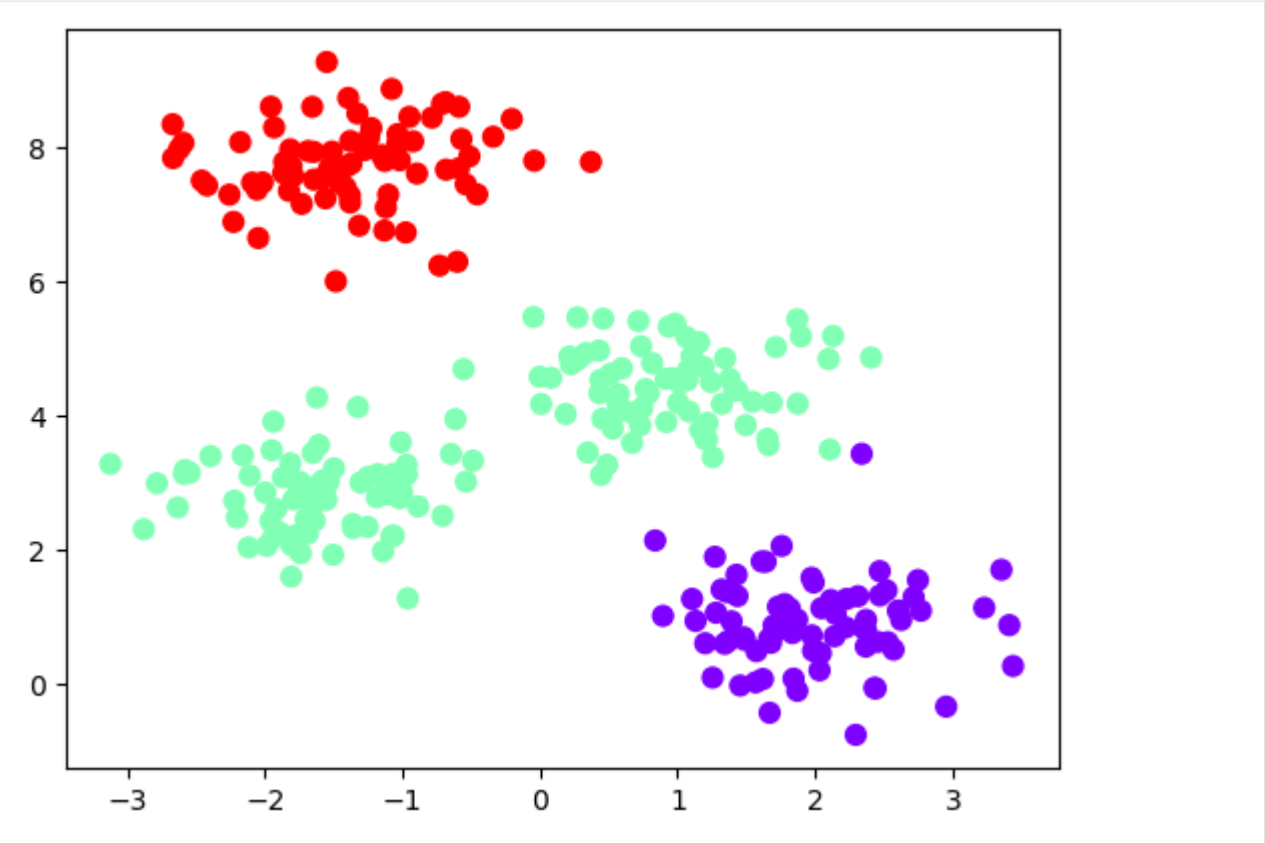
[86]:
```
from sklearn.datasets import make_blobs
# n_samples is number of blobs, centers is number of clusters, cluster_std is the␣
↪standard deviation of each cluster
X3, y3 = make_blobs(n_samples=300, centers=4, random_state=0, cluster_std=0.60)
plt.scatter(X3[:, 0], X3[:, 1], s=50);
```



[87]:
```
k_means_2 = KMeans(n_clusters=4, random_state=0)  # 4 clusters
k_means_2.fit(X3)
kmeans_pred = k_means_2.predict(X3)
plt.scatter(X3[:, 0], X3[:, 1], c=kmeans_pred, s=50, cmap='rainbow');
```



[88]:
```
k_means_2 = KMeans(n_clusters=3, random_state=0)  # 3 clusters
k_means_2.fit(X3)
kmeans_pred = k_means_2.predict(X3)
plt.scatter(X3[:, 0], X3[:, 1], c=kmeans_pred, s=50, cmap='rainbow');
```



7.13.5 13.5 Hyperparameter Tuning

During model training, the **parameters** of the model are updated in an iterative process.

Hyperparameters (tuning parameters) are a set of parameters that control the complexity of the model. For instance, in k-Nearest Neighbors, the number of nearest neighbors is a hyperparameter of the model. Unlike the model parameters that are learned during training, hyperparameters are selected (i.e., tuned) by the user. While models such as kNN have only one or two hyperparameters, other models such as deep neural networks can have many.

Hyperparameter tuning is the process of screening hyperparameter values (or combinations of hyperparameter values) to find a model that generalizes well to unseen data. The performance of some models (e.g., deep neural networks) can significantly depend on the value of hyperparameters.

For instance, to find a suitable value for the number of nearest neighbors in kNN, we can write a for-loop to train the kNN model multiple times and evaluate the performance when the number of nearest neighbors varies from 3 to 15, as in the next cell. We can see that the best performance is achieved for 3 to 5 neighbors.

[89]:
```
from sklearn.neighbors import KNeighborsClassifier

for n_neighbors in range(3, 15):
    knn_model = KNeighborsClassifier(n_neighbors)

    knn_model.fit(X_train, y_train)

    accuracy = knn_model.score(X_test, y_test)
    print('Number of neighbors is {0:}, the test accuracy is {1:5.2f} %'.format(n_
↪neighbors, accuracy*100))
```

(continues on next page)

(continued from previous page)

```
Number of neighbors is 3, the test accuracy is 97.37 %
Number of neighbors is 4, the test accuracy is 97.37 %
Number of neighbors is 5, the test accuracy is 97.37 %
Number of neighbors is 6, the test accuracy is 94.74 %
Number of neighbors is 7, the test accuracy is 94.74 %
Number of neighbors is 8, the test accuracy is 94.74 %
Number of neighbors is 9, the test accuracy is 94.74 %
Number of neighbors is 10, the test accuracy is 92.11 %
Number of neighbors is 11, the test accuracy is 94.74 %
Number of neighbors is 12, the test accuracy is 94.74 %
Number of neighbors is 13, the test accuracy is 94.74 %
Number of neighbors is 14, the test accuracy is 94.74 %
```

13.5.1 Grid Search

Scikit-learn offers the function `GridSearchCV()` for hyperparameter tuning, which searches through different values of hyperparameters to find the best combination of values based on a used performance metric. The letters CV in the name of the function stand for cross-validation, which is explained in the next section.

When there are several hyperparameters to tune, doing it manually can take time and resources, and functions like GridSearchCV can help to automate the tuning of hyperparameters.

In the cell below, we can see that `GridSearchCV()` takes as arguments the model, a grid of hyperparameter values named `hyper_grid`, and a scoring function (in this case `accuracy`). It will loop through the predefined values for the hyperparameters in the grid and fit the model for each predefined value. Based on the scoring function, it will find the best values from the listed hyperparameters.

[90]:
```
from sklearn.model_selection import GridSearchCV

knn_model = KNeighborsClassifier()

# Create grid of hyperparameter values
hyper_grid = {'n_neighbors': [3, 6, 9, 12]}

# Tune a knn model using grid search
grid_search = GridSearchCV(knn_model, hyper_grid, scoring='accuracy')
results = grid_search.fit(X_train, y_train)
```

The best accuracy score on the training dataset and best value of the hyperparameters are attributes of the `results` from the grid search.

[91]:
```
results.best_score_
```

[91]:
```
0.9557312252964426
```

[92]:
```
results.best_params_
```

[92]:
```
{'n_neighbors': 3}
```

And, we can check the accuracy on the test set as well to confirm that the results are consistent with the accuracies in the previous section.

[93]:
```
accuracy = grid_search.score(X_test, y_test)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))
```

```
The test accuracy is 97.37 %
```

One more example is shown below, where a Support Vector Machines Classifier is used with the same dataset, and grid search is performed over two hyperparameters: `C` is a regularization value, and `kernel` is the kernel type. The function searches over all combinations of the two hyperparameters, and the best accuracy is achieved for `C=0.1` and `kernel='linear'`.

[94]:
```
from sklearn.svm import SVC

svc_model = SVC()

hyper_grid = {'C': [0.01, 0.1, 1.0, 10.0, 100.0],
              'kernel': ['linear', 'rbf', 'poly']}

grid_search = GridSearchCV(svc_model, hyper_grid, scoring='accuracy')
results = grid_search.fit(X_train, y_train)

print('Accuracy:',results.best_score_)
print('Hyperparameters:', results.best_params_)
```

```
Accuracy: 0.9739130434782609
Hyperparameters: {'C': 1.0, 'kernel': 'poly'}
```

[95]:
```
accuracy = grid_search.score(X_test, y_test)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))
```

```
The test accuracy is 97.37 %
```

13.5.2 Random Search

However, a Cartesian grid search approach performed by `GridSeachCV()` has limitations, because it does not scale well when the number of hyperparameters to tune is high. **Random search** based on hyperparameter distributions is another approach for hyperparameter tuning which has often demonstrated better performance than grid search.

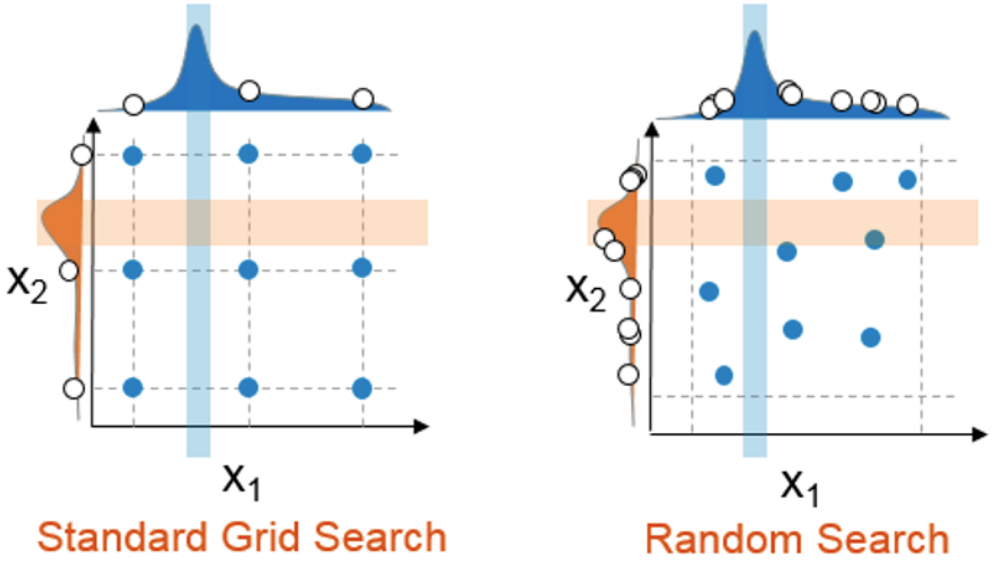


Figure source: Reference [4].

An example of using Random Search with the SVM classifier is shown next where we used the function `RandomizedSearchCV()`. Unlike the grid search, in this case, the distribution of the hyperparameter C is specified, as a uniform distribution with values between 0.01 and 100. Also, in the `RandomizedSearchCV()` we need to provide the number of samples for the hyperparameter from the distribution via the `n_iter` argument.

[96]:
```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform, randint

svc_model = SVC()

# specify hyperparameter distributions to randomly sample from
hyper_distributions = {'C': uniform(0.01, 100)}

random_search = RandomizedSearchCV(svc_model, hyper_distributions, n_iter=40, scoring=
↪'accuracy')
results = random_search.fit(X_train, y_train)

print('Accuracy:',results.best_score_)
print('Hyperparameters:', results.best_params_)
```

```
Accuracy: 0.9735177865612649
Hyperparameters: {'C': 29.838232595603078}
```

[97]:
```
accuracy = random_search.score(X_test, y_test)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))
```

```
The test accuracy is 100.00 %
```

One more example of a random search is shown with a Random Forest classifier. Random Forests can have several hyperparameters, and here we considered the number of trees in the forest with `n_estimators`, maximum depth of the trees is `max_depth`, and the minimum number of samples required in a leaf node is `min_samples_leaf`. These hyperparameters are sampled from a uniform distribution of integers, with the ranges of values defined for each hyperparameter using the function `randint()`. For this case, a standard grid search would be more computationally expensive.

[98]:
```
from sklearn.ensemble import RandomForestClassifier

rf_model = RandomForestClassifier(random_state=123)

# specify hyperparameter distributions to randomly sample from
hyper_distributions = {
    'n_estimators': randint(20, 100),
    'max_depth': randint(4, 20),
    'min_samples_leaf': randint(1, 100),
}

random_search_2 = RandomizedSearchCV(rf_model, hyper_distributions, n_iter=20, scoring=
↪'accuracy')
results = random_search_2.fit(X_train, y_train)

print('Accuracy:',results.best_score_)
print('Hyperparameters:', results.best_params_)
```

```
Accuracy: 0.9466403162055336
Hyperparameters: {'max_depth': 10, 'min_samples_leaf': 10, 'n_estimators': 80}
```

[99]:
```
accuracy = random_search_2.score(X_test, y_test)
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))
```

```
The test accuracy is 97.37 %
```

7.13.6 13.6 Cross-Validation

So far, we adopted a strategy to split our data into training and testing sets, and we assessed the performance of our model on the test set. Unfortunately, there are a few pitfalls to this approach:

1. If our dataset is small, a single test set may not provide an adequate measure of the model's performance on unseen data.
2. A single test set does not provide insights regarding the variability of the model's performance.

An alternative strategy is to use a resampling method, where we can repeatedly fit a model to parts of the training data and test its performance on other parts of the training data. This allows us to train and validate our model entirely on the training data and not touch the test data until we have selected a final "optimal" model. The two most commonly used resampling methods include **k-fold cross-validation** and **bootstrap sampling**. This section focuses on k-fold cross-validation.

13.6.1 K-Fold Cross-Validation

Cross-validation involves repeating the training and testing procedure so that the training and testing sets are different each time. The values of the performance metrics are collected for each repetition and then aggregated. This allows to obtain an estimate of the variability of the model's performance.

k-fold cross-validation is a resampling method that randomly divides the training data into k groups (referred to as *folds*) of approximately equal size.

First, the model is fit on $k-1$ folds and then the remaining fold is used to compute the model's performance. This procedure is repeated k times; each time, a different fold is treated as the validation set. This process results in k estimates of the accuracy, and the overall accuracy is calculated by averaging the k estimates.

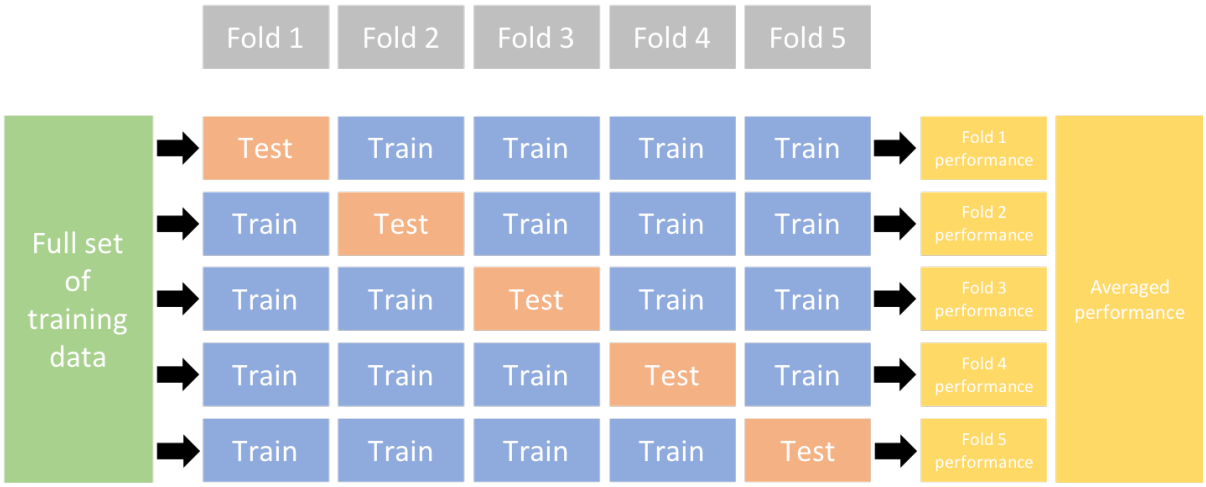


Figure source: Reference [4].

In scikit-learn, the function `cross_validate` allows us to perform cross-validation, and we need to pass to it the model, the data, the target, and the number of folds with `cv`.

In practice, it is most common to use `k=5` or `k=10`.

[100]:
```
from sklearn.model_selection import cross_validate

rf_model = RandomForestClassifier(n_estimators=56, max_depth=9, min_samples_leaf=3,␣
↪random_state=123)

cv_result = cross_validate(rf_model, X_train, y_train, cv=5)
cv_result
```

[100]:
```
{'fit_time': array([0.05299497, 0.05299735, 0.09700108, 0.39899945, 0.17099905]),
 'score_time': array([0.00500154, 0.00400066, 0.04000068, 0.03599882, 0.0049994 ]),
 'test_score': array([0.86956522, 1.        , 0.86363636, 0.95454545, 1.        ])}
```

The output of `cross_validate` is a Python dictionary (named `cv_rsults` in our case), which by default contains three entries:

- `fit_time`: the time to train the model on the training data for each fold.
- `score_time`: the time to predict with the model on the testing data for each fold.
- `test_score`: the score (such as accuracy) on the testing data for each fold.

In the next cell, we displayed the mean and standard deviation of the accuracy for the five folds, by using the values for the `'test_score'` key in the resulting dictionary.

[101]:
```
scores = cv_result["test_score"]
print("The mean cross-validation accuracy is: "
      f"{scores.mean():.3f} +/- {scores.std():.3f}")
```

```
The mean cross-validation accuracy is: 0.938 +/- 0.060
```

And optionally, we can manually perform k-fold cross-validation by using a for-loop and feed the folds for training and testing. In the code below `train` and `test` are the indices of the elements in each fold.

[102]:
```
from sklearn.model_selection import StratifiedKFold
```

(continues on next page)

(continued from previous page)

```
kfold = StratifiedKFold(n_splits=5).split(X_train, y_train)

scores = []
for k, (train, test) in enumerate(kfold):
    rf_model.fit(X_train[train], y_train[train])
    accuracy_score = rf_model.score(X_train[test], y_train[test])
    scores.append(accuracy_score)
    print('Fold: %2d, Class dist.: %s, Acc: %.3f' % (k+1,
          np.bincount(y_train[train]), accuracy_score))

print('\nCV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```

```
Fold:  1, Class dist.: [30 30 29], Acc: 0.870
Fold:  2, Class dist.: [30 30 29], Acc: 1.000
Fold:  3, Class dist.: [30 30 30], Acc: 0.864
Fold:  4, Class dist.: [31 29 30], Acc: 0.955
Fold:  5, Class dist.: [31 29 30], Acc: 1.000

CV accuracy: 0.938 +/- 0.060
```

7.13.7 13.7 Performance Metrics

In the above examples, we worked only with accuracy as a performance metric, but other metrics that are also frequently used with prediction models.

We will use the breast cancer dataset to explain the metrics. The dataset consists of diagnoses for 569 patients, and includes 30 features for each patient, which are shown below. The targets are the benign or malignant class, shown in the column `target` to the right.

[103]:
```
import pandas as pd
from sklearn.datasets import load_breast_cancer

bc = load_breast_cancer()
df = pd.DataFrame(data=bc.data, columns=bc.feature_names)
df["target"] = bc.target
df.head()
```

[103]:
```
   mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0        17.99         10.38          122.80     1001.0          0.11840
1        20.57         17.77          132.90     1326.0          0.08474
2        19.69         21.25          130.00     1203.0          0.10960
3        11.42         20.38           77.58      386.1          0.14250
4        20.29         14.34          135.10     1297.0          0.10030

   mean compactness  mean concavity  mean concave points  mean symmetry  \
0           0.27760          0.3001              0.14710         0.2419
1           0.07864          0.0869              0.07017         0.1812
2           0.15990          0.1974              0.12790         0.2069
3           0.28390          0.2414              0.10520         0.2597
4           0.13280          0.1980              0.10430         0.1809

   mean fractal dimension  ...  worst texture  worst perimeter  worst area  \
```

(continues on next page)

(continued from previous page)

```
0                 0.07871  ...          17.33           184.60      2019.0
1                 0.05667  ...          23.41           158.80      1956.0
2                 0.05999  ...          25.53           152.50      1709.0
3                 0.09744  ...          26.50            98.87       567.7
4                 0.05883  ...          16.67           152.20      1575.0

   worst smoothness  worst compactness  worst concavity  worst concave points  \
0            0.1622             0.6656           0.7119                0.2654
1            0.1238             0.1866           0.2416                0.1860
2            0.1444             0.4245           0.4504                0.2430
3            0.2098             0.8663           0.6869                0.2575
4            0.1374             0.2050           0.4000                0.1625

   worst symmetry  worst fractal dimension  target
0          0.4601                  0.11890       0
1          0.2750                  0.08902       0
2          0.3613                  0.08758       0
3          0.6638                  0.17300       0
4          0.2364                  0.07678       0

[5 rows x 31 columns]
```

[104]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 31 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   mean radius              569 non-null    float64
 1   mean texture             569 non-null    float64
 2   mean perimeter           569 non-null    float64
 3   mean area                569 non-null    float64
 4   mean smoothness          569 non-null    float64
 5   mean compactness         569 non-null    float64
 6   mean concavity           569 non-null    float64
 7   mean concave points      569 non-null    float64
 8   mean symmetry            569 non-null    float64
 9   mean fractal dimension   569 non-null    float64
 10  radius error             569 non-null    float64
 11  texture error            569 non-null    float64
 12  perimeter error          569 non-null    float64
 13  area error               569 non-null    float64
 14  smoothness error         569 non-null    float64
 15  compactness error        569 non-null    float64
 16  concavity error          569 non-null    float64
 17  concave points error     569 non-null    float64
 18  symmetry error           569 non-null    float64
 19  fractal dimension error  569 non-null    float64
 20  worst radius             569 non-null    float64
 21  worst texture            569 non-null    float64
 22  worst perimeter          569 non-null    float64
```

(continues on next page)

(continued from previous page)

```
 23  worst area               569 non-null    float64
 24  worst smoothness         569 non-null    float64
 25  worst compactness        569 non-null    float64
 26  worst concavity          569 non-null    float64
 27  worst concave points     569 non-null    float64
 28  worst symmetry           569 non-null    float64
 29  worst fractal dimension  569 non-null    float64
 30  target                   569 non-null    int32
dtypes: float64(30), int32(1)
memory usage: 135.7 KB
```

[105]:
```
df['target'].value_counts()
```

[105]:
```
1    357
0    212
Name: target, dtype: int64
```

There are 357 patients with malignant tumors, and 212 patients with benign tumors. Let's create train and test datasets.

[106]:
```
X = df.drop('target', axis=1)
y = df['target']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0,␣
↪stratify=y)
```

[107]:
```
print('Training data inputs', X_train.shape)
print('Training labels', y_train.shape)
print('Testing data inputs', X_test.shape)
print('Testing labels', y_test.shape)
```

```
Training data inputs (398, 30)
Training labels (398,)
Testing data inputs (171, 30)
Testing labels (171,)
```

A Logistic Regression model achieved 92.98% accuracy.

[108]:
```
lr_model = LogisticRegression()

# fit model
lr_model.fit(X_train, y_train)

# accuracy
lr_model.score(X_test, y_test)
```

[108]:
```
0.935672514619883
```

[109]:
```
from sklearn.metrics import confusion_matrix

y_pred = lr_model.predict(X_test)

confmat = confusion_matrix(y_test, y_pred, labels=[1, 0])
print(confmat)
```

```
[[101   6]
 [  5  59]]
```

The confusion matrix can be interpreted as the values of positives and negatives, if there are only two classes (binary classification). For the case of breast cancer they mean:

- True Positive (TP): Predicted malignant, actually malignant (positive). There are 101 TP samples in this case.
- True Negative (TN): Predicted benign, actually benign (negative). There are 59 TN samples in this case.
- False Negative (FN): Predicted benign, actually malignant (positive). There are 6 FN samples in this case.
- False Positive (FP): Predicted malignant, actually benign (negative). There are 5 FP samples in this case.

Obviously, correctly guessing TP and TN is the goal. Other than that, FN is the worst case, because it means that a malignant tumor is missed. FP is also bad, because it will require to perform biopsy and will put stress on the patient, but the outcome is still preferred.

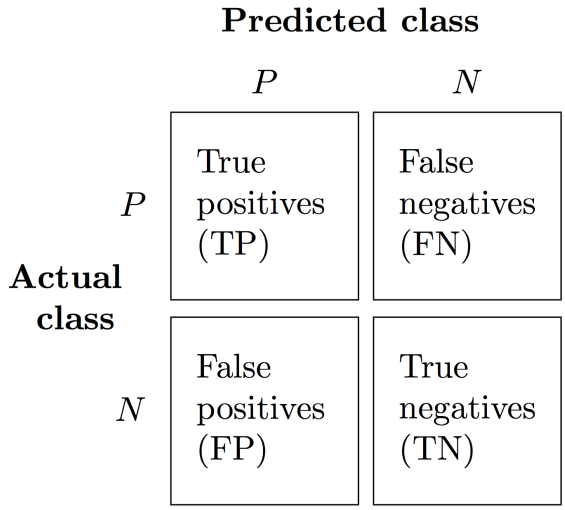


Figure source: Reference [2].

The following performance metrics are commonly used.

Accuracy = $(TP + TN)/(TP + FN + TN + FP)$, or it is the fraction of correct predictions by the model among all available samples. In this case it is $(101 + 59)/(101 + 6 + 5 + 59) = 160/171 = 0.9357$.

Precision = $TP/(TP + FP)$, or it is the fraction of correct positive predictions by the model among all positive predictions by the model. In this case it is $101/(101 + 5) = 101/106 = 0.9528$. Or, you can think of precision as, from all patients that the model predicted to have malignant tumors, how many patients have malignant tumors.

Recall = $TP/(TP + FN)$, or it is the fraction of all positive predictions by the model from all positive samples in the data. In this case it is $101/(101 + 6) = 101/107 = 0.9439$. Or, you can think of recall as, from all patients that have malignant tumors, how many patients the model predicted that have malignant tumors. In binary classification, recall is also called **sensitivity** or **true positive rate**.

Specificity = $TN/(TN + FP)$, or it is the fraction of all negative predictions by the model from all negative samples in the data. In this case it is $59/(59 + 5) = 59/64 = 0.9219$. Or, you can think of specificity as, from all patients that have benign tumors, how many patients the model predicted that have benign tumors. Specificity is also called **true negative rate**.

F1 score = $(2 * precision * recall)/(precision + recall)$, is a measure which combines the precision and recall, and it is also called *harmonic mean of precision and recall*. F1 score is considered as a more suitable metric than accuracy in cases when the dataset is unbalanced, that is, there are more samples from one class than the other class (or classes).

$$
\begin{aligned}
precision &= \frac{TP}{TP + FP} \\
recall &= \frac{TP}{TP + FN} \\
F1 &= \frac{2 \times precision \times recall}{precision + recall} \\
accuracy &= \frac{TP + TN}{TP + FN + TN + FP} \\
specificity &= \frac{TN}{TN + FP}
\end{aligned}
$$

In scikit-learn, we can obtain the individual metrics as follows.

[110]:
```
from sklearn import metrics

print(metrics.precision_score(y_test, y_pred))
print(metrics.recall_score(y_test, y_pred))
print(metrics.f1_score(y_test, y_pred))
```

```
0.9528301886792453
0.9439252336448598
0.948356807511737
```

Also, the function `classification_report` prints the precision, recall, f1-score, and accuracy for all classes. The reported averages include **macro average accuracy** (averaging the accuracy per each class), and **weighted average** (averaging the accuracy by using weights for each class based on the proportions of samples in each class). The suuport column shows the number of samples, i.e., rom the 171 samples in the test dataset, there are 64 benign and 107 malignant cases.

[111]:
```
# multiple metrics at once
print(metrics.classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.91      0.92      0.91        64
           1       0.95      0.94      0.95       107

    accuracy                           0.94       171
   macro avg       0.93      0.93      0.93       171
weighted avg       0.94      0.94      0.94       171
```

And one more commonly used metric in binary classification is **Receiver Operating Characteristic (ROC)**. ROC curves are created by plotting the False Positive Rate (= $FP/(FP + TN)$) on the x-axis and True Positive Rate (= $TP/(TP + FN)$, or sensitivity, racall) on the y-axis. The curve summarizes the trade-off between the True Positive Rate and False Positive Rate of the model using different probability thresholds.

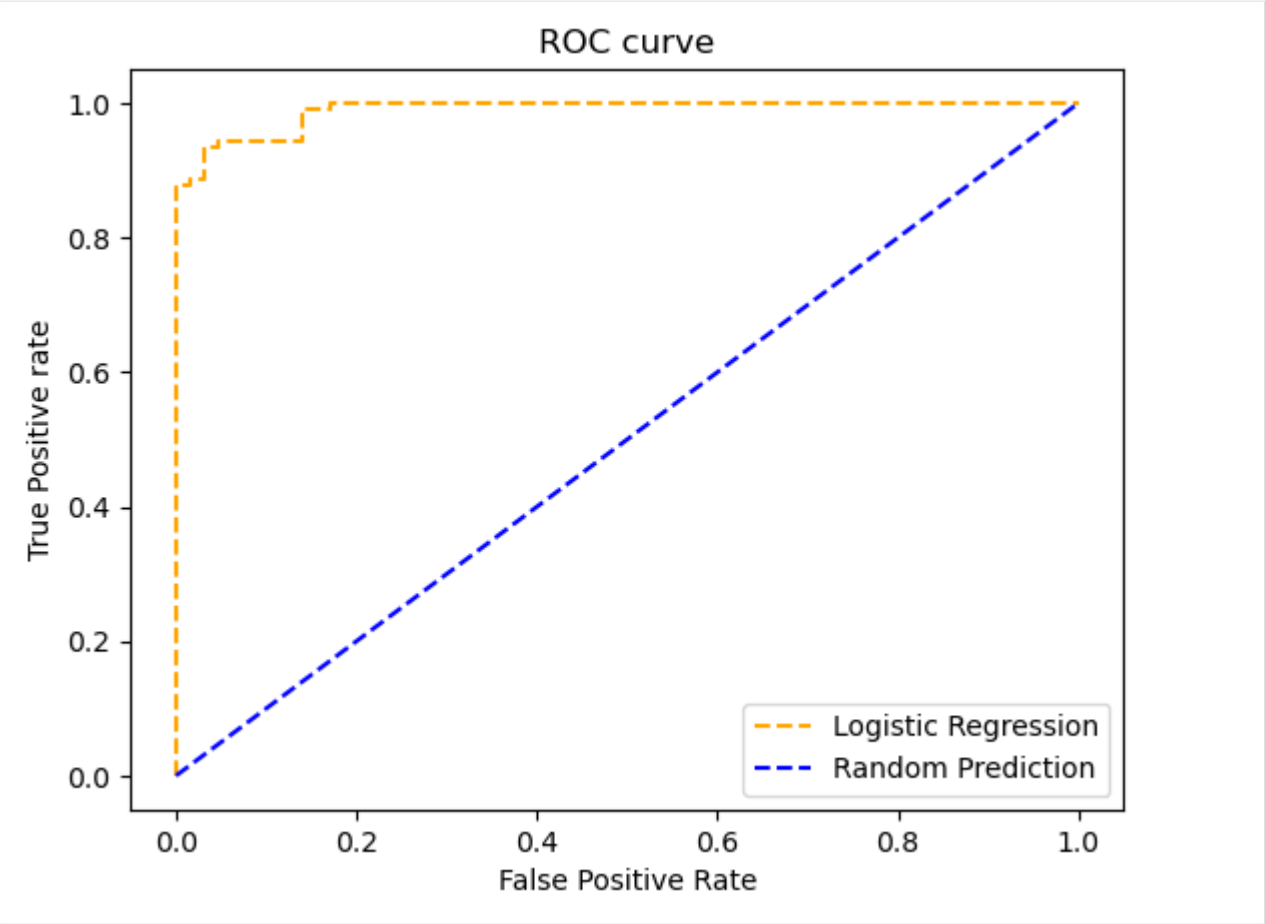
So far, we only used a threshold of 0.5, where predictions greater or equal to it are assigned the class label 1, and predictions smaller than 0.5 are assigned the class label 0. In binary classification, using different thresholds for the probability in predicting the class label of a data point changes the True Positive Rate and False Positive Rate. By changing the threshold value from 0 to 1 we can obtain many different points with True Positive Rate and False Positive Rate, and by connecting the different points we can plot the ROC curve. The curve allows to better interpret the results of the classifier.

In a ROC curve, the top left corner of the plot is the ideal point - with a False Positive Rate of 0, and a True Positive Rate of 1. In general, a higher x-value indicates higher number of False Positives than True Negatives, and a higher y-value indicates higher number of True Positives than False Negatives.

The plot below shows the ROC curve for the Logistic Regression model. It also shows a line for a model that makes random predictions. The further the curve from the line with random prediction, the better the model predictions are.

[112]:
```
# output probabilities for each sample
y_pred_probabilities = lr_model.predict_proba(X_test)
```

[113]:
```
from sklearn.metrics import roc_curve

# output probabilities for each sample
y_pred_probabilities = lr_model.predict_proba(X_test)

# calculate FPR and TRP
fpr, tpr, _ = roc_curve(y_test, y_pred_probabilities[:,1])

# roc curve for tpr = fpr
random_probs = [0 for i in range(len(y_test))]
p_fpr, p_tpr, _ = roc_curve(y_test, random_probs)

plt.plot(fpr, tpr, linestyle='--', color='orange', label='Logistic Regression')
plt.plot(p_fpr, p_tpr, linestyle='--', color='blue', label='Random Prediction')
plt.title('ROC curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive rate')
plt.legend(loc='best')
plt.show()
```



The **Area Under the Curve (AUC) of ROC (AUC-ROC)** is the measure of the ability of a classifier to distinguish between classes and is used as a summary of the ROC curve. The higher the AUC, the better the performance of the model at distinguishing between the positive and negative classes. When AUC = 1, the classifier is able to perfectly predict the classes. When AUC = 0.5, the classifier is making random predictions of the class for the data points.

In scikit-learn, AUC-ROC is calculated using the `auc_roc_score`.

[114]:
```
metrics.roc_auc_score(y_test, y_pred)
```

[114]:
```
0.9329001168224299
```

7.13.8 13.8 Model Pipelines

We can combine sequential operations with a scikit-learn `Pipeline`, which chains together operations. The helper function `make_pipeline` is used to create a `Pipeline`, and takes as arguments the successive transformations to perform, followed by the classifier or regressor model.

Let's create one pipeline consisting of a Standard Scaler function and a Logistic Regression classifier, and another pipeline consisting of a MinMax Scaler function and a Random Forest classifier.

[115]:
```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler, MinMaxScaler
```

(continues on next page)

(continued from previous page)

```
pipeline_ss_lr = make_pipeline(StandardScaler(), LogisticRegression())
pipeline_ns_rf = make_pipeline(MinMaxScaler(), RandomForestClassifier())
```

Apply the first pipeline to perform data scaling via standardization and afterward fit the Logistic Regression model to the data.

[116]:
```
# apply the pipeline
pipeline_ss_lr.fit(X_train, y_train)

# score on the test data
accuracy = pipeline_ss_lr.score(X_test, y_test)
print('The test accuracy of Logistic Regression is {0:5.2f} %'.format(accuracy*100))
```

```
The test accuracy of Logistic Regression is 95.91 %
```

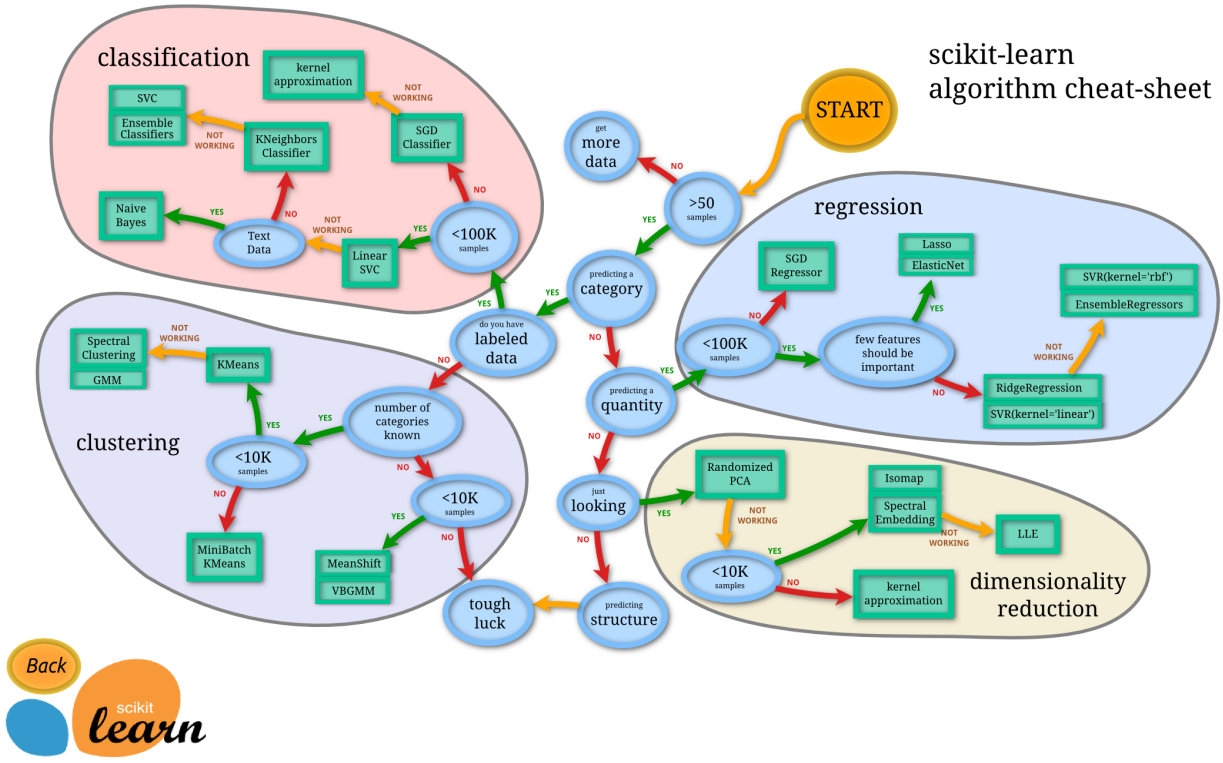
Apply the second pipeline to perform data scaling via normalization and afterward fit the Random Forest model to the data.

[117]:
```
# apply the pipeline
pipeline_ns_rf.fit(X_train, y_train)

# score on the test data
accuracy = pipeline_ns_rf.score(X_test, y_test)
print('The test accuracy of Random Forests is {0:5.2f} %'.format(accuracy*100))
```

```
The test accuracy of Random Forests is 93.57 %
```

7.13.9 13.9 Flow Chart: How to Choose an Estimator

This is a flow chart created by the scikit-learn contributor Andreas Mueller that gives a nice summary of which algorithms to choose in various situations.



Recap: Scikit-learn's Estimator Interface

Scikit-learn strives to have a uniform interface across all methods, as we saw in the examples. Given a scikit-learn *estimator* object named `model`, the following methods are available:

- Available in **all estimators**
  - `model.fit()`: fit training data. For supervised learning applications, this accepts two arguments: the data `X` and the labels `y` (e.g. `model.fit(X, y)`). For unsupervised learning applications, this accepts only a single argument, the data `X` (e.g. `model.fit(X)`).
- Available in **supervised estimators**
  - `model.predict()`: given a trained model, predict the label of a new set of data. This method accepts one argument, the new data `X_new` (e.g. `model.predict(X_new)`), and returns the learned label for each item in the array.
  - `model.predict_proba()`: For classification tasks, some estimators also provide this method, which returns the probability that a new sample has for each category. In this case, the label with the highest probability is returned by `model.predict()`.
  - `model.score()`: for classification or regression tasks, most estimators implement a score method. Scores are between 0 and 1, with a larger score indicating a better fit.
- Available in **unsupervised estimators**
  - `model.predict()`: predict labels in clustering algorithms.
  - `model.transform()`: given an unsupervised model, transform new data into a new basis. This also accepts one argument `X_new`, and returns the new representation of the data based on the unsupervised model.

- `model.fit_transform()`: some estimators implement this method, which more efficiently performs a fit and a transform on the same input data.

7.13.10 Appendix

The material in the Appendix is not required for quizzes and assignments.

Application Example: Optical Character Recognition

To demonstrate the above principles on a more interesting problem, let's consider OCR (Optical Character Recognition) – that is, recognizing hand-written digits. Here, we will use scikit-learn's set of pre-formatted digits, which can be loaded directly with `load_digits()`.

Loading and visualizing the digits data

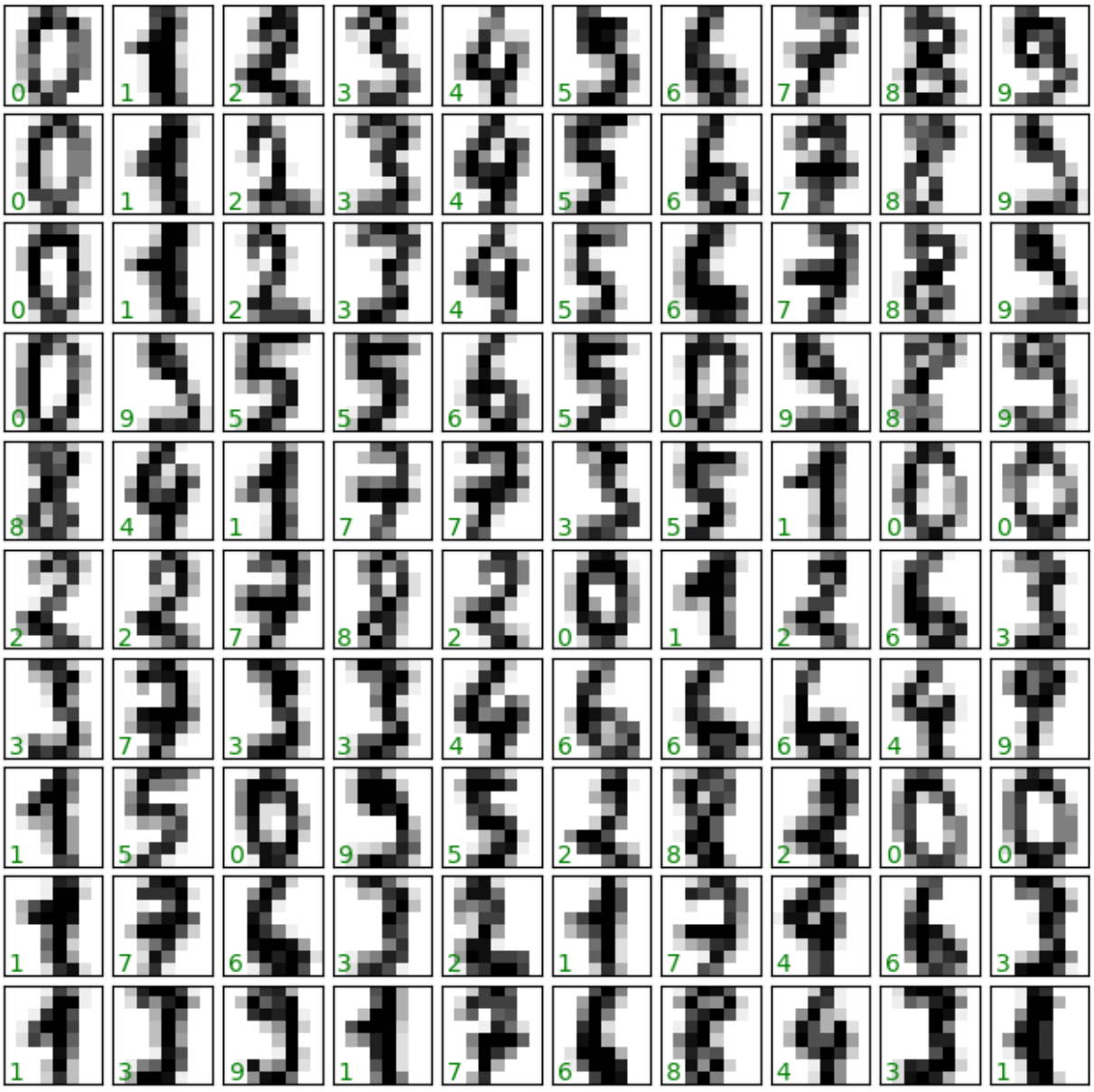
We'll use scikit-learn's data access interface and take a look at this data:

[118]:
```
from sklearn import datasets
digits = datasets.load_digits()
digits.images.shape
```

[118]:
```
(1797, 8, 8)
```

Let's plot a few images and show the ground-truth label in the lower left corner.

[119]:
```
import numpy as np
import matplotlib.pyplot as plt

fig, axes = plt.subplots(10, 10, figsize=(8, 8))
fig.subplots_adjust(hspace=0.1, wspace=0.1)

for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[i], cmap='binary')
    ax.text(0.05, 0.05, str(digits.target[i]),
            transform=ax.transAxes, color='green')
    ax.set_xticks([])
    ax.set_yticks([])
```



The images are just 8x8 pixels.

[120]:
```
# The images themselves
print(digits.images.shape)
print(digits.images[0])
```

```
(1797, 8, 8)
[[ 0.  0.  5. 13.  9.  1.  0.  0.]
 [ 0.  0. 13. 15. 10. 15.  5.  0.]
 [ 0.  3. 15.  2.  0. 11.  8.  0.]
 [ 0.  4. 12.  0.  0.  8.  8.  0.]
 [ 0.  5.  8.  0.  0.  9.  8.  0.]
 [ 0.  4. 11.  0.  1. 12.  7.  0.]
```

(continues on next page)

(continued from previous page)

```
 [ 0.  2. 14.  5. 10. 12.  0.  0.]
 [ 0.  0.  6. 13. 10.  0.  0.  0.]]
```

Because scikit-learn uses two-dimensional inputs, the images has been reshaped from 8x8 pixels into one long vector of 64 elements.

[121]:
```
# The data for use in our algorithms
print(digits.data.shape)
print(digits.data[0])
```

```
(1797, 64)
[ 0.  0.  5. 13.  9.  1.  0.  0.  0.  0. 13. 15. 10. 15.  5.  0.  0.  3.
 15.  2.  0. 11.  8.  0.  0.  4. 12.  0.  0.  8.  8.  0.  0.  5.  8.  0.
  0.  9.  8.  0.  0.  4. 11.  0.  1. 12.  7.  0.  0.  2. 14.  5. 10. 12.
  0.  0.  0.  0.  6. 13. 10.  0.  0.  0.]
```

[122]:
```
# The target label
print(digits.target)
```

```
[0 1 2 ... 8 9 8]
```

The data have 1797 samples having 64 dimensions (features).

Classification on Digits

Let's first split the digits into a training and testing sample.

[123]:
```
Xtrain, Xtest, ytrain, ytest = train_test_split(digits.data, digits.target, random_
↪state=2)
print(Xtrain.shape, Xtest.shape)
```

```
(1347, 64) (450, 64)
```

And, we will use a simple Logistic Regression as a classification algorithm.

[124]:
```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(penalty='l2')
clf.fit(Xtrain, ytrain)
ypred = clf.predict(Xtest)
```

[125]:
```
from sklearn.metrics import accuracy_score
accuracy_score(ytest, ypred)
```

[125]:
```
0.9466666666666667
```

[126]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(ytest, ypred))
```

```
[[41  0  0  0  1  0  0  0  0  0]
 [ 0 46  0  1  0  0  0  0  2  1]
 [ 0  0 47  0  0  0  0  0  0  0]
 [ 0  0  0 44  0  1  0  1  2  0]
 [ 0  0  0  0 36  0  0  0  3  1]
```

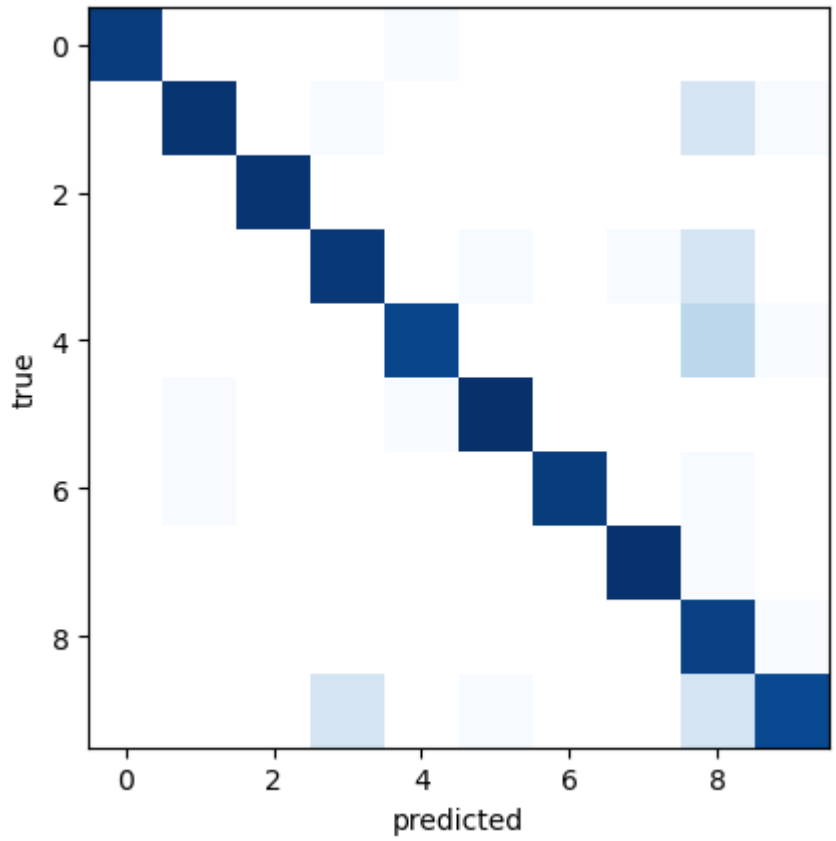
(continues on next page)

(continued from previous page)

```
 [ 0  1  0  0  1 50  0  0  0  0]
 [ 0  1  0  0  0  0 41  0  1  0]
 [ 0  0  0  0  0  0  0 48  1  0]
 [ 0  0  0  0  0  0  0  0 39  1]
 [ 0  0  0  2  0  1  0  0  2 34]]
```

We can plot the confusion matrix as an image.

[127]:
```
plt.imshow(np.log(confusion_matrix(ytest, ypred)),
           cmap='Blues', interpolation='nearest')
plt.grid(False)
plt.ylabel('true')
plt.xlabel('predicted');
```



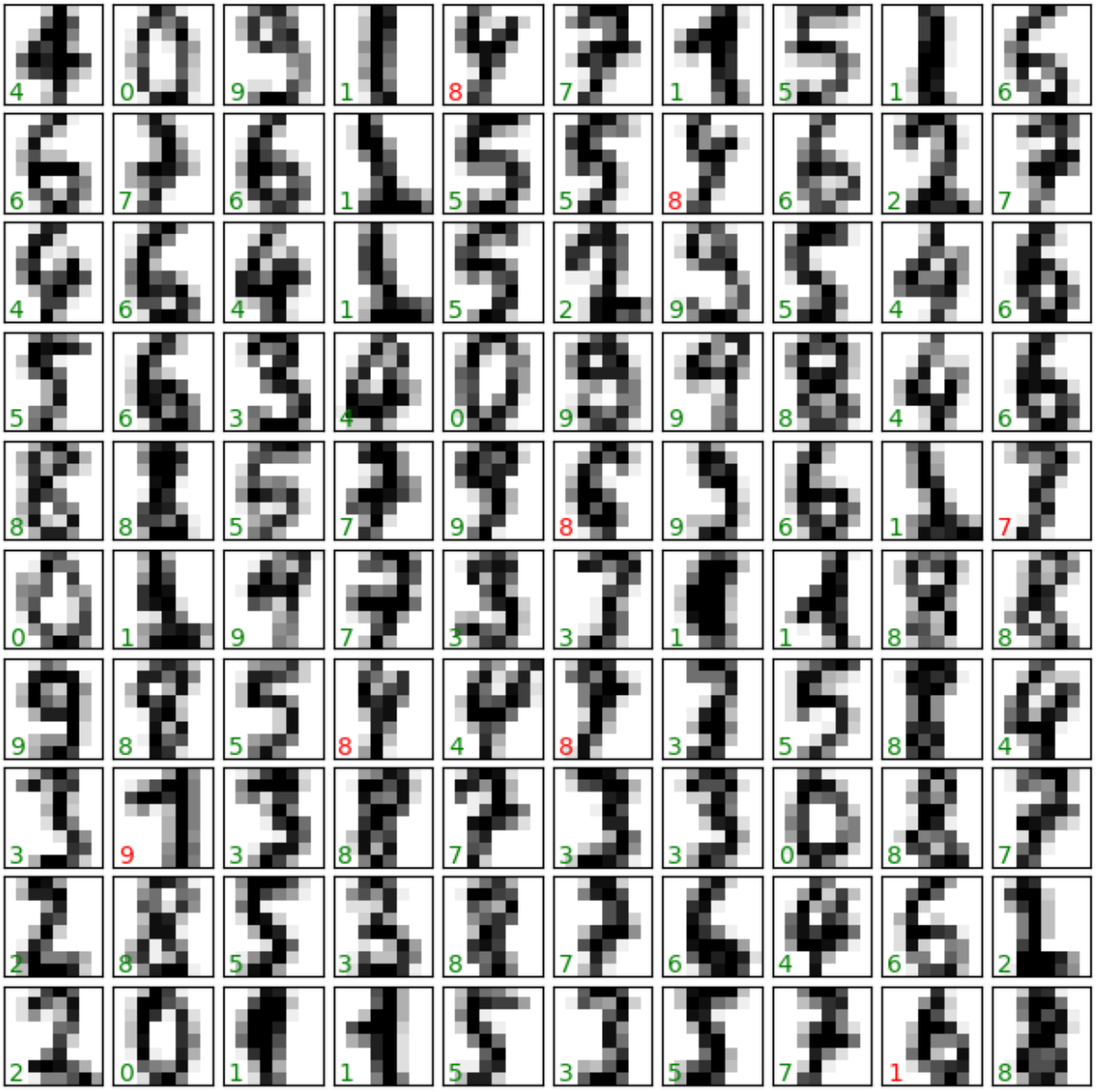
We can also look at some of the images along with their predicted labels. The incorrect labels are in red color.

[128]:
```
fig, axes = plt.subplots(10, 10, figsize=(8, 8))
fig.subplots_adjust(hspace=0.1, wspace=0.1)

for i, ax in enumerate(axes.flat):
    ax.imshow(Xtest[i].reshape(8, 8), cmap='binary')
    ax.text(0.05, 0.05, str(ypred[i]),
            transform=ax.transAxes,
            color='green' if (ytest[i] == ypred[i]) else 'red')
    ax.set_xticks([])
```

(continues on next page)

(continued from previous page)

```
    ax.set_yticks([])
```



In fact, some of the mislabeled cases with this simple logistic regression algorithm are difficult to classify even for us.

7.13.11 References

1. PyCon 2015 Sckikit-Learn Tutorial, Jake VanderPlas available at: https://github.com/jakevdp/sklearn_pycon2015.
2. Python Machine Learning (2nd Ed.) Code Repository, Sebastian Raschka, available at: https://github.com/rasbt/python-machine-learning-book-2nd-edition.
3. Complete Machine Learning Package, Jean de Dieu Nyandwi, available at: https://github.com/Nyandwi/machine_learning_complete.
4. Advanced Python for Data Science, University of Cincinnati, available at: https://github.com/uc-python/advanced-python-datasci.
5. Scikit-Learn: Machine Learning Concepts, available at: (https://inria.github.io/scikit-learn-mooc/toc.html) {[}https://inria.github.io/scikit-learn-mooc/toc.html].

BACK TO TOP

7.14 Lecture 14 - Ensemble Methods

7.14.1 14.1 Ensemble Methods

Ensemble methods combine the predictions of several other machine learning models, where by aggregating the results of the other models trained on the same dataset, ensembles can typically achieve better performance than any individual model. The individual models are often referred to as *base models* or *base learners*.

When we introduced scikit-learn we mentioned that Random Forest is an example of an ensemble model, since it combines a multitude of Decision Tree models to make predictions.

Ensembles are one of the most powerful Machine Learning methods, and they are often used in the winning solutions in many Machine Learning competitions.

Ensemble methods can be used for different learning tasks, including classification and regression. In this lecture, we will focus on ensemble classifiers.

Ensemble models are classified into four general groups:

- **Voting Methods**: make predictions based on majority voting of the individual models.
- **Bagging Methods**: train individual models on random subsets of the training data.
- **Boosting Methods**: train individual models sequentially by learning from the errors. Examples are Gradient Boosting, XGBoost, LightGBM, CatBoost, and AdaBoost.
- **Stacking Methods**: train individual models, and use another model to aggregate their predictions.

Most of the above ensemble methods are implemented in scikit-learn, except for XGBoost, LightGBM, and CatBoost.

14.1.1 Loading the Dataset

To demonstrate the working principles of ensemble methods, we will use a dataset of electricity usage in New South Wales, Australia. In that market, electricity prices are not fixed and they are set every five minutes based on demand and supply of the market.

We will train classifiers to predict whether the electricity price will go UP or DOWN the next time the price is set.

The dataset contains 45,312 rows of data from May 7, 1996 to December 5, 1998. Each row of the dataset refers to a period of 30 minutes. It has information about the date and day of the week, and has other features related to electricity demand, schedule transfer, etc.

Let's load the dataset `electricity_data` using pandas.

[1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import sklearn
import matplotlib.pyplot as plt

# Let's hide warning messages in the cells
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
elec_df = pd.read_csv('data/electricity_data.csv')
```

Let's inspect the dataset. The `date` column does not show the actual date correctly, but we will use it as is, because the date has already been encoded into an adequate numerical format.

The `class` column contains the target labels for the classification task, where we would like to predict whether the electricity price will go UP or DOWN.

[3]:
```
elec_df.head(10)
```

[3]:

	date	day	period	nswprice	nswdemand	vicprice	vicdemand	transfer	\
0	0.0	2	0.000000	0.056443	0.439155	0.003467	0.422915	0.414912	
1	0.0	2	0.021277	0.051699	0.415055	0.003467	0.422915	0.414912	
2	0.0	2	0.042553	0.051489	0.385004	0.003467	0.422915	0.414912	
3	0.0	2	0.063830	0.045485	0.314639	0.003467	0.422915	0.414912	
4	0.0	2	0.085106	0.042482	0.251116	0.003467	0.422915	0.414912	
5	0.0	2	0.106383	0.041161	0.207528	0.003467	0.422915	0.414912	
6	0.0	2	0.127660	0.041161	0.171824	0.003467	0.422915	0.414912	

(continues on next page)

(continued from previous page)

```
7   0.0    2  0.148936  0.041161   0.152782  0.003467   0.422915  0.414912
8   0.0    2  0.170213  0.041161   0.134930  0.003467   0.422915  0.414912
9   0.0    2  0.191489  0.041161   0.140583  0.003467   0.422915  0.414912

  class
0    UP
1    UP
2    UP
3    UP
4  DOWN
5  DOWN
6  DOWN
7  DOWN
8  DOWN
9  DOWN
```

Exploratory Data Analysis

[4]:
```
elec_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45312 entries, 0 to 45311
Data columns (total 9 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   date       45312 non-null  float64
 1   day        45312 non-null  int64
 2   period     45312 non-null  float64
 3   nswprice   45312 non-null  float64
 4   nswdemand  45312 non-null  float64
 5   vicprice   45312 non-null  float64
 6   vicdemand  45312 non-null  float64
 7   transfer   45312 non-null  float64
 8   class      45312 non-null  object
dtypes: float64(7), int64(1), object(1)
memory usage: 3.1+ MB
```

From the summary statistics below, we can tell that the maximum value in each column is 1 and the minimum is 0, meaning that the features are already normalized, so we don't need to apply data scaling. This does not apply to the day column that has values between 1 and 7 (i.e., Monday to Sunday), but we can leave that column as is.

[5]:
```
# summary statistics
elec_df.describe()
```

[5]:
```
                date           day        period      nswprice     nswdemand  \
count  45312.000000  45312.000000  45312.000000  45312.000000  45312.000000
mean       0.499080      4.003178      0.500000      0.057868      0.425418
std        0.340308      1.998695      0.294756      0.039991      0.163323
min        0.000000      1.000000      0.000000      0.000000      0.000000
25%        0.031934      2.000000      0.250000      0.035127      0.309134
50%        0.456329      4.000000      0.500000      0.048652      0.443693
```

(continues on next page)

(continued from previous page)

```
75%         0.880547      6.000000      0.750000      0.074336      0.536001
max         1.000000      7.000000      1.000000      1.000000      1.000000

           vicprice     vicdemand      transfer
count  45312.000000  45312.000000  45312.000000
mean       0.003467      0.422915      0.500526
std        0.010213      0.120965      0.153373
min        0.000000      0.000000      0.000000
25%        0.002277      0.372346      0.414912
50%        0.003467      0.422915      0.414912
75%        0.003467      0.469252      0.605702
max        1.000000      1.000000      1.000000
```

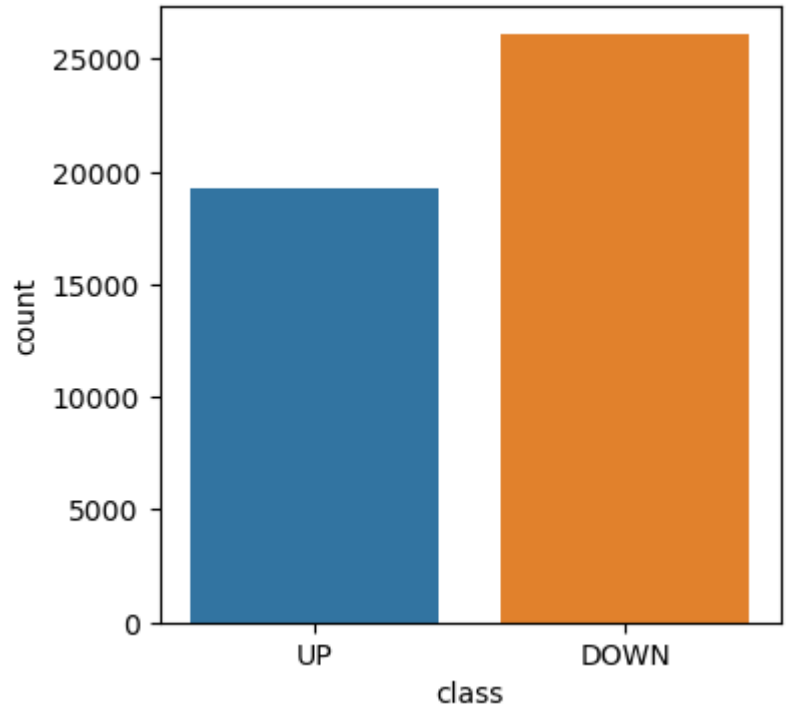
[6]:
```
# Checking missing values
elec_df.isnull().sum()
```

[6]:
```
date         0
day          0
period       0
nswprice     0
nswdemand    0
vicprice     0
vicdemand    0
transfer     0
class        0
dtype: int64
```

Let's see how many UPs and DOWNs are in the `class` column.

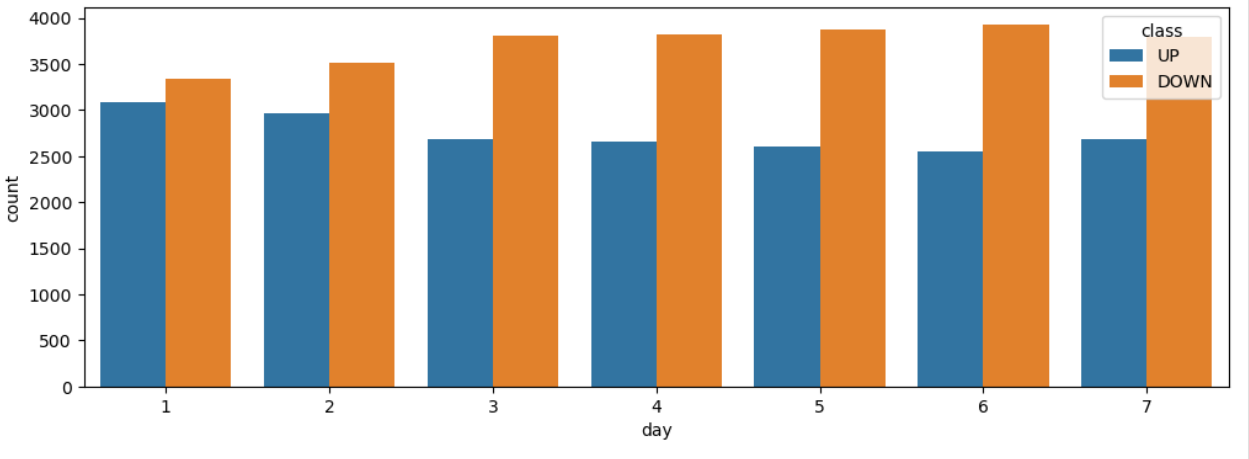
[7]:
```
plt.figure(figsize=(4,4))
sns.countplot(data=elec_df, x='class')
```

[7]:
```
<Axes: xlabel='class', ylabel='count'>
```



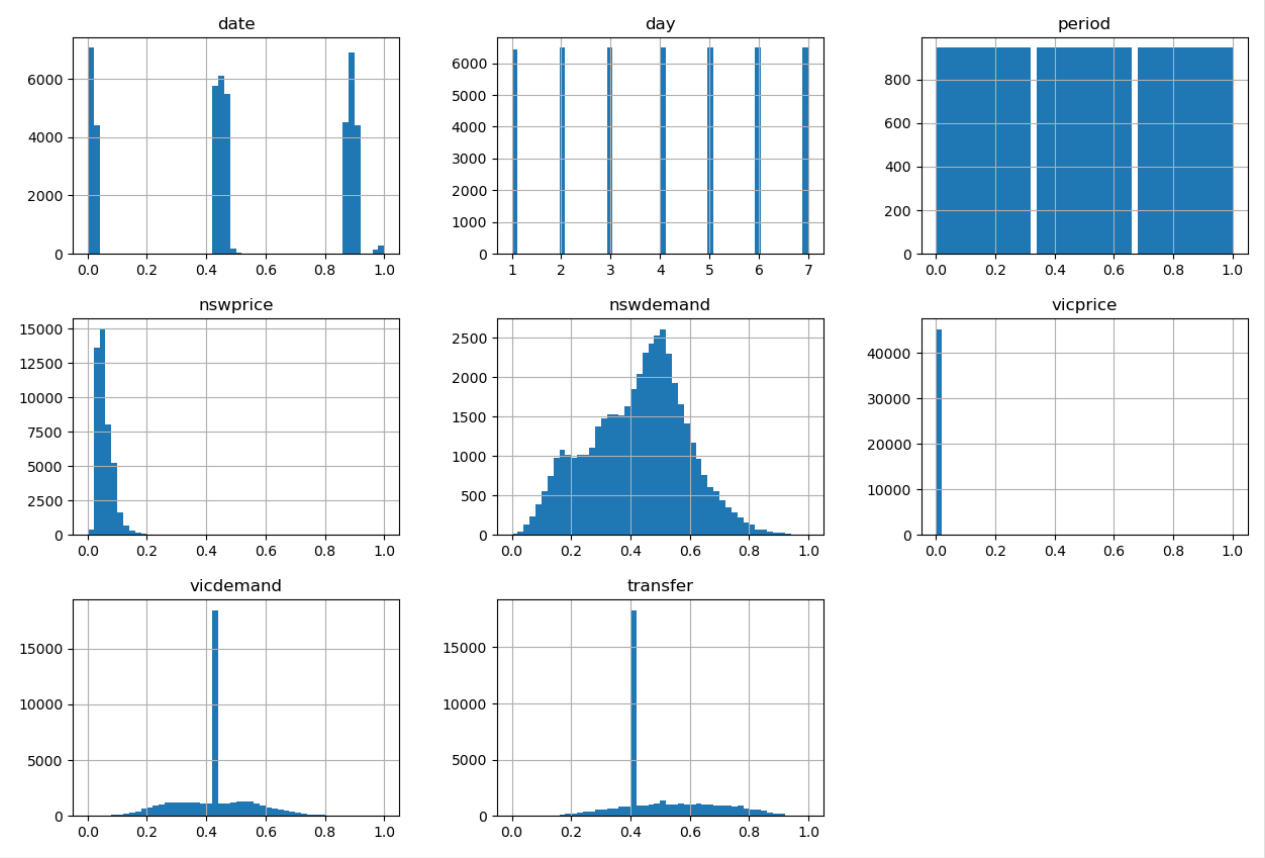
Let's also inspect the UPs and DOWNs of the electricity price for each day of the week.

[8]:
```
plt.figure(figsize=(12,4))
sns.countplot(data=elec_df, x='day', hue='class')
plt.show()
```



We can also plot all histograms of the numerical features, and quickly check the plots.

[9]:
```
elec_df.hist(bins=50, figsize=(15,10))
plt.show()
```



Data Preprocessing

We will assign the data to the variable `X` and the targets (the column `class`) to `y`.

Recall again from the lecture on Data Exploration and Preprocessing that if the dataset contained an `index` column it would be important to remove that column from the training dataset `X`, to avoid associating the target labels with the indices of the rows by the model. This dataset does not have an `index` column, therefore we don't need to worry about that.

[10]:
```
X = elec_df.drop('class', axis=1)
y = elec_df['class']
```

Since the target feature `class` is categorical, let's encode it to numerical values using Label Encoder.

[11]:
```
from sklearn.preprocessing import LabelEncoder

label_enc = LabelEncoder()
y = label_enc.fit_transform(y)
```

[12]:
```
y
```

[12]:
```
array([1, 1, 1, ..., 0, 1, 0])
```

And let's split the data into training and test sets.

[13]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_
↪state=20)
```

[14]:
```
print('Training data inputs', X_train.shape)
print('Training labels', y_train.shape)
print('Testing data inputs', X_test.shape)
print('Testing labels', y_test.shape)
```

```
Training data inputs (33984, 8)
Training labels (33984,)
Testing data inputs (11328, 8)
Testing labels (11328,)
```

As we mentioned previously, the data values are already scaled to the range [0, 1], therefore we don't need to apply feature scaling. If that was not the case, we would have applied one of the Scaler functions in scikit-learn.

7.14.2 14.2 Voting Ensemble

Voting ensemble is the simplest ensemble technique, which aggregates the predictions of several individual classifiers and assigns the predicted class labels based on the majority votes of the single classifiers.

For example, if we have a dataset with two classes A and B, and we train three classifiers. Then, for one particular test sample:

- Classifier 1 predicts class A
- Classifier 2 predicts class B
- Classifier 3 predicts class B
- Ensemble majority voting is: class B (2 votes), class A (1 vote). The prediction by the Ensemble Method is: class B.

Let's first train 3 classifiers on the training data for `electricity_data`, including: Logistic Regression, Support Vector Machines, and SGD. Afterward, we will apply a voting ensemble that uses their results to make predictions.

[15]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

lr_model = LogisticRegression()

# train
lr_model.fit(X_train, y_train)

# evaluate
lr_preds = lr_model.predict(X_test)
lr_acc = accuracy_score(y_test, lr_preds)
print('Logistic Regression accuracy is {0:7.4f} %'.format(lr_acc*100))
```

```
Logistic Regression accuracy is 75.5297 %
```

[16]:
```
from sklearn.svm import SVC
```

(continues on next page)

(continued from previous page)

```
svm_model = SVC(random_state=1)

# train
svm_model.fit(X_train, y_train)

# evaluate
svm_preds = svm_model.predict(X_test)
svm_acc = accuracy_score(y_test, svm_preds)
print('SVM accuracy is {0:7.4f} %'.format(svm_acc*100))
```

```
SVM accuracy is 73.6935 %
```

[17]:
```
from sklearn.linear_model import SGDClassifier

sgd_model = SGDClassifier(random_state=1)

# train
sgd_model.fit(X_train, y_train)

# evaluate
sgd_preds = sgd_model.predict(X_test)
sgd_acc = accuracy_score(y_test, sgd_preds)
print('SGD accuracy is {0:7.4f} %'.format(sgd_acc*100))
```

```
SGD accuracy is 75.3708 %
```

[18]:
```
print('Accuracy: Logistic Regression {0:7.4f}; SVM {1:7.4f}; SGD {2:7.4f}'.format(lr_
↪acc*100, svm_acc*100, sgd_acc*100))
```

```
Accuracy: Logistic Regression 75.5297; SVM 73.6935; SGD 75.3708
```

In scikit-learn we will import `Voting Classifier` to aggregate the results of those 3 classifiers. In the cell below, the models are listed in the `estimators` argument of the `Voting Classifier`, and afterward the `Voting Classifier` model is fit on the training data. The voting argument `hard` means that the ensemble algorithm uses the predicted class labels for majority voting.

[19]:
```
from sklearn.ensemble import VotingClassifier

voting_classifier = VotingClassifier(

    estimators=[('log_reg', lr_model),
                ('svc', svm_model),
                ('sgd', sgd_model)],
    voting='hard')

voting_classifier.fit(X_train, y_train)
```

[19]:
```
VotingClassifier(estimators=[('log_reg', LogisticRegression()),
                             ('svc', SVC(random_state=1)),
                             ('sgd', SGDClassifier(random_state=1))])
```

[20]:
```
# evaluate
voting_preds = voting_classifier.predict(X_test)
```

(continues on next page)

(continued from previous page)

```
voting_acc = accuracy_score(y_test, voting_preds)
print('Voting Ensemble accuracy is {0:7.4f} %'.format(voting_acc*100))
```

```
Voting Ensemble accuracy is 75.6532 %
```

The Voting Ensemble achieved slightly higher accuracy than the three models.

Beside `hard` voting, another alternative for the voting ensemble classifier is to change the argument to `voting='soft'`. In this case, the decision by the ensemble is made based on the average of the predicted class with the highest probabilities of all individual classifiers.

Example of soft voting:

- Classifier 1 predicts class A with probability 90%, class B with probability 10%
- Classifier 2 predicts class A with probability 45%, class B with probability 55%
- Classifier 3 predicts class A with probability 45%, class B with probability 55%
- Ensemble probabilities: class A ((90% + 45% + 45%) / 3 = 60%), class B ((10% + 55% + 55%) / 3 = 40%). Prediction: class A.

Note in the above example that if hard voting based on the class labels was applied, the prediction would have been class B, because both Classifiers 2 and 3 predicted higher probabilities for class B.

Soft voting generally achieves better performance than hard voting, because it assigns higher weight to the predictions with high confidence by the individual models.

In the cell below, to apply soft voting with SVC we need to change the argument `probability=True`, as well as this approach requires to modify the outputs of the SGD model, and instead of SGD we will use a k-Nearest Neighbors model.

[21]:
```
from sklearn import neighbors

knn_model = neighbors.KNeighborsClassifier()
svm_classifier = SVC(gamma="auto", probability=True)

voting_classifier_soft = VotingClassifier(

    estimators=[('log_reg', lr_model),
                ('svc', svm_classifier),
                 ('knn', knn_model)],
     voting='soft')

voting_classifier_soft.fit(X_train, y_train)
```

[21]:
```
VotingClassifier(estimators=[('log_reg', LogisticRegression()),
                             ('svc', SVC(gamma='auto', probability=True)),
                             ('knn', KNeighborsClassifier())],
                 voting='soft')
```

[22]:
```
# evaluate
voting_preds = voting_classifier_soft.predict(X_test)
voting_acc = accuracy_score(y_test, voting_preds)
print('Voting Ensemble accuracy is {0:7.4f} %'.format(voting_acc*100))
```

```
Voting Ensemble accuracy is 80.7998 %
```

Ensemble Methods are generally well-performing. Even in the cases when the individual models are *weak learners* (which means that they predict slightly better than random guessing), ensemble learning can result in a *strong learner* (meaning that it has high prediction accuracy), as long as a large number of diverse individual models are used.

In general, ensemble methods produce the best results when the individual models are independent of each other, e.g., they use different types of learning algorithms. This way, the individual models can produce diverse results, which can increase the chance that the ensembling will be more successful. Conversely, if the individual models use the same learning algorithm, they will probably make the same type of errors, which will reduce the accuracy when an ensemble method is used.

Diversity in individual learners can be achieved by selecting models that use different learning algorithms, or by training models on different subsets of the training data, or different data features, etc. Also, many ensemble methods employ Decision Trees as individual models, where diversity can be achieved by varying the strategies for trees spliting, by varying the different hyperparameters, or by using techniques based on data diversity, feature diversity, and other techniques. Specific approaches are explained next.

7.14.3 14.3 Bagging Ensemble

As we mentioned, one way to create diverse base models is to train them on different portions of the training data. Hence, instead of training different models on the same data and averaging their results with a Voting Ensemble, **Bagging Ensemble** method trains the base models on different subsets of the training data and aggregates the results.

The term **bagging** is short for *bootstrap aggregating*, where bootstrapping refers to sampling subsets from the training data with replacement. In other words, the subsets of the training data used by different models can contain the same data points. When the subsets are sampled without replacement, it is called *pasting*.

The aggregation of the predictions by the individual models is typically based on the most frequent prediction, similar to hard voting. The individual models used with bagging are typically Decision Trees.

In scikit-learn we can import a Bagging Classifier, and fit it to the training data as any other model. Let's use bagging to train Decision Trees on different subsets of the data and then average the predictions. In the cell below, `max_samples=0.5` means that each model will use a random subset containing 50% of the training data, and `max_features=0.5` means that each model will use a random subset of 50% of the input features. If bootstrap is `True` the training samples will be sampled with replacement, and if it is `False` there is no replacement (i.e., pasting).

[23]:
```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bagging_classifier = BaggingClassifier(
      DecisionTreeClassifier(class_weight='balanced'),
    max_samples=0.5, max_features=0.5, bootstrap=True
)

bagging_classifier.fit(X_train, y_train)
```

[23]:
```
BaggingClassifier(estimator=DecisionTreeClassifier(class_weight='balanced'),
                  max_features=0.5, max_samples=0.5)
```

[24]:
```
# evaluate
bagging_preds = bagging_classifier.predict(X_test)
bagging_acc = accuracy_score(y_test, bagging_preds)
print('Bagging Ensemble accuracy is {0:7.4f} %'.format(bagging_acc*100))
```

```
Bagging Ensemble accuracy is 83.6335 %
```

Bagging Ensembles often outperform Voting Ensembles, and they can decrease the overfitting of Decision Trees (which tend to overfit easily). As well as, Bagging Ensembles can be trained in parallel using different CPU cores, which can reduce the processing time.

Random Forest can be considered a Bagging Ensemble with the `max_samples` set to 1, that is, it uses the entire training set to train the based learners.

7.14.4 14.4 Boosting Ensemble

Boosting Ensemble trains the individual models sequentially, where each next model has access to the predictions of the previous models and attempts to improve the performance. The term *boosting* refers to incremental improvement by reducing the errors of the predecessors, in order to gradually improve the performance.

The most popular boosting ensembles are Gradient Boosting, XGBoost, LightGBM, CatBoost, and AdaBoost.

14.4.1 Gradient Boosting Ensemble

Gradient Boosting Ensemble fits the individual models sequentially, where each new model is fit to minimize a loss function based on the error (gradient) of a loss function made by the previous models. The individual models in gradient boosting are Decision Trees.

Specifically, the initial model is trained on the full dataset, and the errors (gradients) of the loss function are obtained based on the difference between the predicted class of the model and the target label. The next model is trained by using the negative gradient of the loss function in an attempt to correct the mistakes made by the previous model. These steps are repeated for each single model. Ideally, as more models are trained the errors should become smaller, that is, the predictions by the models should better match the target labels.

The residual errors represent the gradient of the loss function with respect to the predicted values. This is the reason why the method is called Gradient Boosting.

In scikit-learn, Gradient Boosting Classifier has hyperparameters related to the number of estimators i.e. Decision Trees (`n_estimators`), learning rate (`learning_rate`), maximum depth of the trees (`max_depth`), and others. The learning rate is typically set to values between 0.01 and 1, and it defines how quickly the models are updated. If the learning rate is too high, the models will be updated more quickly, which can cause overfitting.

Decison Trees are often used as based models in Ensemble Methods. The reasons are because they are easily diversified by varying the splitting of the trees or by other means, they can capture complex nonlinear relationships in the data, they are fast and simple to implement, and can also provide information about the feature importance.

[25]:
```
from sklearn.ensemble import GradientBoostingClassifier

grad_boost_classifier = GradientBoostingClassifier(
                        n_estimators=500,
                        learning_rate=0.8,
                        random_state=42,
                        max_depth=2)

grad_boost_classifier.fit(X_train, y_train)
```

[25]:
```
GradientBoostingClassifier(learning_rate=0.8, max_depth=2, n_estimators=500,
                           random_state=42)
```

[26]:
```
# evaluate
gboost_preds = grad_boost_classifier.predict(X_test)
```

(continues on next page)

(continued from previous page)

```
gboost_acc = accuracy_score(y_test, gboost_preds)
print('Gradient Boosting Ensemble accuracy is {0:7.4f} %'.format(gboost_acc*100))
```

```
Gradient Boosting Ensemble accuracy is 89.6893 %
```

14.4.2 Recent Gradient Boosting Methods: XGBoost, LightGBM, CatBoost

There are several optimized variants of Gradient Boosting Ensembles, which include XGBoost, LightGBM, and CatBoost. These three ensemble methods are among the most powerful and robust Machine Learning models for tabular data at present time. Their performance is often superior to other conventional ML methods, including other ensemble methods. Also, they are very fast and scalable to large datasets. Because of these properties, XGBoost, LightGBM, and CatBoost are often the winning algorithms in many Kaggle competitions and other ML competitions where the datasets are in tabular format.

Note that XGBoost, LightGBM, and Catboost are not implemented in Scikit-Learn, and they need to be installed as separate libraries, e.g., via:

```
pip install xgboost catboost lightgbm
```

On the other hand, the API for working with these methods is the same as scikit-learn, and they use the `fit` and `predict` methods for model training and evaluation.

14.4.2.1 XGBoost (Extreme Gradient Boosting)

XGBoost or **eXtreme Gradient Boosting Ensemble** introduces several modifications and optimizations in the original Gradient Boosting algorithm, which speed up the training process and improve the predictive performance. The main modifications in XGBoost encompass the introduction of L1 and L2 regularization terms into the objective function to reduce overfitting, and using a more efficient tree-building algorithm that relies on histogram-based methods and pruning techniques to optimize the tree splitting. XGBoost can also handle missing data, either by treating the data samples with missing values as a separate category, or it can apply techniques for filling in the missing values. Therefore, it can work without significant exploratory data analysis and preprocessing.

As shown in the next cells, applying these variants of Gradient Boosting is as simple as fitting and evaluating other scikit-learn models.

[27]:
```
import xgboost as xgb

xgb_classifier = xgb.XGBClassifier()
xgb_classifier.fit(X_train, y_train)
```

[27]:
```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None, random_state=None, ...)
```

[28]:
```
# evaluate
xgboost_preds = xgb_classifier.predict(X_test)
xgboost_acc = accuracy_score(y_test, xgboost_preds)
print('XGBoost Ensemble accuracy is {0:7.4f} %'.format(xgboost_acc*100))
```

```
XGBoost Ensemble accuracy is 90.1660 %
```

14.4.2.2 LightGBM (Light Gradient Boosting Machine)

LightGBM or **Light Gradient Boosting Machine** is another optimized version of Gradient Boosting Ensemble, which was developed by Microsoft. It uses a histogram-based approach for finding optimal splitting of the decision trees. As well as, LightGBM employs a leaf-wise tree growth strategy that can result in deeper trees compared to other ensemble algorithms. LightGBM focuses on efficiency and scalability of the approach.

[29]:
```
import lightgbm
from lightgbm.sklearn import LGBMClassifier

lightgbm_classifier = LGBMClassifier()
lightgbm_classifier.fit(X_train, y_train)
```

```
[LightGBM] [Info] Number of positive: 14443, number of negative: 19541
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.
↪001957 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 1586
[LightGBM] [Info] Number of data points in the train set: 33984, number of used features:
↪ 8
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.424994 -> initscore=-0.302305
[LightGBM] [Info] Start training from score -0.302305
```

[29]:
```
LGBMClassifier()
```

[30]:
```
# evaluate
lightgbm_preds = lightgbm_classifier.predict(X_test)
lightgbm_acc = accuracy_score(y_test, lightgbm_preds)
print('XGBoost Ensemble accuracy is {0:7.4f} %'.format(lightgbm_acc*100))
```

```
XGBoost Ensemble accuracy is 87.9237 %
```

Optimizing the hyperparameters of ensemble methods can result in significant improvement in performance. The next cells demonstrate this, by training another LightGBM model with different hyperparameters.

[31]:
```
lightgbm_classifier_2 = LGBMClassifier(num_leaves=100, learning_rate=0.1, n_
↪estimators=100)
lightgbm_classifier_2.fit(X_train, y_train)
```

```
[LightGBM] [Info] Number of positive: 14443, number of negative: 19541
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.
↪002078 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 1586
[LightGBM] [Info] Number of data points in the train set: 33984, number of used features:
↪ 8
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.424994 -> initscore=-0.302305
```

(continues on next page)

(continued from previous page)

```
[LightGBM] [Info] Start training from score -0.302305
```

[31]:
```
LGBMClassifier(num_leaves=100)
```

[32]:
```
# evaluate
lightgbm_2_preds = lightgbm_classifier_2.predict(X_test)
lightgbm_2_acc = accuracy_score(y_test, lightgbm_2_preds)
print('XGBoost Ensemble accuracy is {0:7.4f} %'.format(lightgbm_2_acc*100))
```

```
XGBoost Ensemble accuracy is 91.6314 %
```

In general, tuning the hyperparameters of Gradient Boosting variants can be non-trivial, and requires to get familiar with the various hyperparameters and understand their impact on the performance.

14.4.2.3 CatBoost (Categorical Boosting)

CatBoost or **Categorical Boosting** is specifically designed for handling categorical features, and it can operate directly on data with categorical features, without the requirement for applying ordinal or label encoding or other preprocessing techniques. It uses an ordered gradient boosting technique for direct processing of categorical features. Other modifications in CatBoost include the use of specialized regularization techniques (such as depth regularization and feature permutation-based regularization) and implementation of specialized symmetric trees structure to prevent overfitting.

Similar to XGBoost, CatBoost has also support for dealing with missing data values, which simplifies the data preprocessing.

[33]:
```
import catboost
from catboost import CatBoostClassifier

catboost_classifier = CatBoostClassifier(verbose=False)
catboost_classifier.fit(X_train, y_train)
```

[33]:
```
<catboost.core.CatBoostClassifier at 0x1bf39c82450>
```

[34]:
```
# evaluate
catboost_preds = catboost_classifier.predict(X_test)
catboost_acc = accuracy_score(y_test, catboost_preds)
print('XGBoost Ensemble accuracy is {0:7.4f} %'.format(catboost_acc*100))
```

```
XGBoost Ensemble accuracy is 89.2744 %
```

The decision on which variant of Gradient Boosting to select depends on the specific characteristics of a dataset, the available computational resources, the type of features in the dataset, and other factors. All three methods are high-performing, fast, and robust, and they are continuously developed and improved.

14.4.3 AdaBoost Ensemble

AdaBoost Ensemble is another class of Boosting Ensembles, however, instead of minimizing the gradient of a loss function as in Gradient Boosting models, it assigns weights to the training data instances based on the difficulty to be classified. The main idea of AdaBoost is to focus the individual models on the training instances that are difficult to classify. AdaBoost is short of *adaptive boosting*.

AdaBoost begins by training a base model (which by default is a Decision Tree) on the full training dataset. After that, higher weights are assigned to the data instances that were misclassified by the previous model. Therefore, the second model will put more attention on correctly predicting the class of the misclassified data instances. By training many models sequentially, the algorithm will try harder to predict the difficult data instances.

The aggregated prediction by all single models is done by averaging the predictions and using weights for each model based on their accuracy on the training set.

The main parameters in the AdaBoost Classifier are the type of base estimators and the number of estimators.

[35]:
```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

adaboost_classifier = AdaBoostClassifier(
    base_estimator=DecisionTreeClassifier(max_depth=3, class_weight='balanced'),
    n_estimators=300,
    learning_rate=0.5
)

adaboost_classifier.fit(X_train, y_train)
```

[35]:
```
AdaBoostClassifier(base_estimator=DecisionTreeClassifier(class_weight='balanced',
                                                         max_depth=3),
                   learning_rate=0.5, n_estimators=300)
```

[36]:
```
# evaluate
adaboost_preds = adaboost_classifier.predict(X_test)
adaboost_acc = accuracy_score(y_test, adaboost_preds)
print('AdaBoost Ensemble accuracy is {0:7.4f} %'.format(adaboost_acc*100))
```

```
AdaBoost Ensemble accuracy is 89.9100 %
```

7.14.5 14.5 Stacking Ensemble

Stacking Ensemble is also referred to as *stacked generalization*, where instead of using hard or soft voting for aggregating the results from multiple models, another model (referred to as a *blender* or *meta-learner*) is trained to perform the aggregation.

In the example below, we train two base estimators: SVM classifier and a Random Forest classifier. Afterward, we use Logistic Regression model as a final estimator, which is trained by using the predictions by the two base estimators as inputs. The final estimator aggregates the predictions of the individual models and outputs the final predictions.

[37]:
```
from sklearn.ensemble import StackingClassifier
from sklearn.ensemble import RandomForestClassifier

base_estimators = [
    ('rand', RandomForestClassifier(random_state=42)),
    ('svc', SVC(random_state=42))]
```

(continues on next page)

(continued from previous page)

```
final_estimator = LogisticRegression()

stack_classifier = StackingClassifier(estimators = base_estimators,
                                      final_estimator = final_estimator)

stack_classifier.fit(X_train, y_train)
```

[37]:
```
StackingClassifier(estimators=[('rand',
                                RandomForestClassifier(random_state=42)),
                               ('svc', SVC(random_state=42))],
                   final_estimator=LogisticRegression())
```

[38]:
```
# evaluate
stack_preds = stack_classifier.predict(X_test)
stack_acc = accuracy_score(y_test, stack_preds)
print('Stacking Ensemble accuracy is {0:7.4f} %'.format(stack_acc*100))
```

```
Stacking Ensemble accuracy is 90.2189 %
```

In conclusion, Ensemble Methods are among the best performing Machine Learning models. Whereas they may not be competitive with Neural Networks for datasets containing images, text, audio, or other data formats, for datasets with tabular data Ensemble Methods may outperform Neural Network-based approaches. In addition, Ensemble Methods can achieve good performance even without hyperparameter finetuning, in comparison to Neural Networks that typically require significant hyperparameter finetuning. As well as, Ensemble Methods can outperform Neural Networks for tasks with smaller dataset sizes, are robust to outliers, and can handle imbalanced datasets.

7.14.6 References

1. Complete Machine Learning Package, Jean de Dieu Nyandwi, available at: https://github.com/Nyandwi/machine_learning_complete.
2. Hands-on Machine Learning with Scikit-learn, Keras & TensorFlow, Aurelien Geron, available at: https://github.com/ageron/handson-ml/blob/master/07_ensemble_learning_and_random_forests.ipynb.
3. Python Machine Learning (2nd Ed.) Code Repository, Sebastian Raschka, available at: https://github.com/rasbt/python-machine-learning-book-2nd-edition.

BACK TO TOP

7.15 Lecture 15 - Artificial Neural Networks

7.15.1 15.1 Introduction to Artificial Neural Networks

Artificial Neural Networks (ANNs) are a class of machine learning algorithms that employ a network of connected computational units for information processing. ANNs are organized in layers, where each layer is made up of interconnected nodes (or computational units). The concept of ANNs is motivated by biological neural networks, and the development of ANNs is the result of an attempt to replicate the workings of the human brain. Hence, the nodes in ANNs are commonly referred to as **neurons**.

A simple ANN with one single layer is depicted in the following figure. The main layers in the ANN are:

- **Input layer**, takes the input data in numerical format, where inputs to ANNs can be tabular data, images, texts, audio, etc.
- **Hidden layer**, connects the input and the output layers. The neurons in the hidden layer process the input data and extract relevant patterns in the data.
- **Output layer**, is the output of the network, which can be a continuous numeric value as in regression problems, or discrete class values as in classification problems.

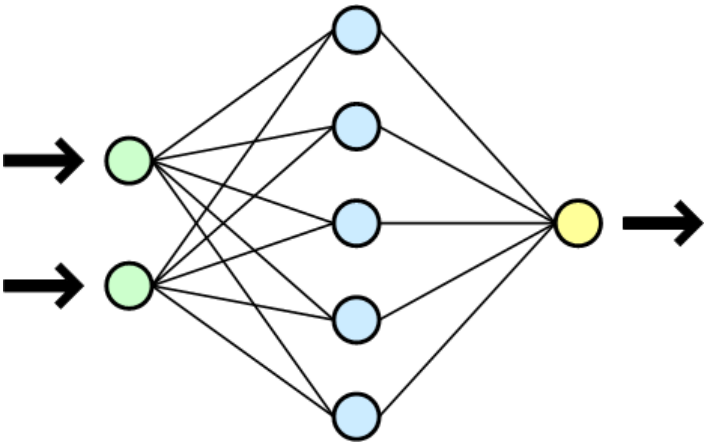


Figure: One-layer ANN. Figure source: Wikipedia.

In general, ANNs can have multiple input layers, multiple hidden layers, and multiple output layers.

ANNs having more than one layer are commonly referred to as a **Deep Neural Network (DNNs)**. Similarly, applying DNNs for solving a task is referred to as **Deep Learning (DL)**.

Most current ANNs that are implemented for solving real-world tasks have multiple hidden layers. The number of hidden layers depends on the specific task, and typically ranges from 10 to 100. For solving complex tasks, ANNs with hundreds of hidden layers are used.

15.1.1 Elements of ANNs

The values of the connections between the neurons in ANNs are learned during a training phase. By using a training dataset, the values of the connections are updated with a goal of correctly predicting the outputs for given inputs.

The following figure depicts the operation of a single neuron in an ANN. The neuron is connected to other neurons in the previous layer via a set of inputs $x_1, x_2, ..., x_m$ and weights $w_1, w_2, ..., w_m$. Each neuron calculates the sum of products of the inputs and weights and a bias value, i.e., $v = x_1 \cdot w_1 + x_2 \cdot w_2 + ... + x_m \cdot w_m + b$. The obtained value v is then passed through an activation function and the output $y = \varphi(v)$ is transferred to the next layer in the network.

The **weights** and **biases** in an ANN are called **network parameters**. And very often the term *weights* is used to refer to both weights and biases as network parameters.

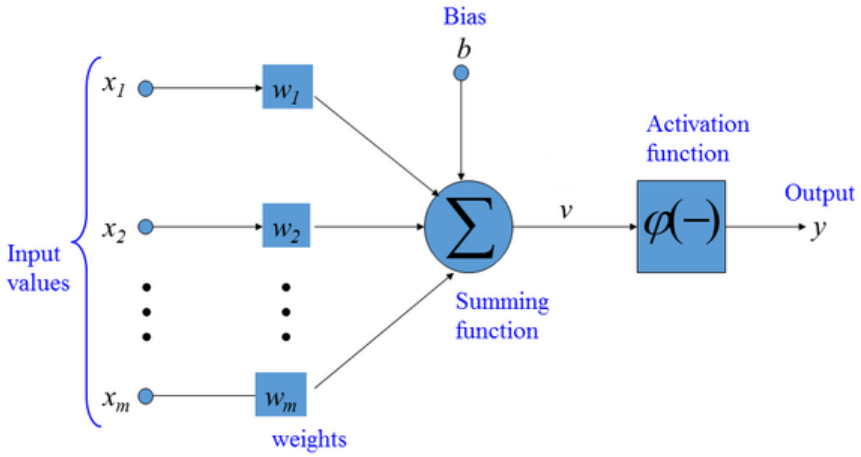


Figure: Artificial Neuron.

15.1.2 Common Activation Functions

Activation functions are applied to each neuron in ANNs to introduce non-linearities into the outputs of the neurons. Without non-linear activation functions, ANNs will simply learn linear mappings between the inputs and outputs. Calculating non-linear mappings between inputs and outputs gives ANNs the capabilities to learn complex tasks.

The most common activation functions in ANNs are:

- **Sigmoid** activation function (logistic activation function): For a real-valued input, it outputs a value between 0 and 1. Sigmoid activation is used in the output layer for binary classification problems.
- **Tanh** activation function (tangent hyperbolic activation function): It is similar to sigmoid activation function, but it outputs a value between -1 and 1. Tanh is rarely used in modern ANNs.
- **Softmax** activation function: It is an extension of the sigmoid activation function for multiclass classification tasks, and it is used in the output layer for multiclass classification problems.
- **ReLU (Rectified Linear Unit)** activation function: It outputs the input value if it is positive, and it outputs 0 if the input value is negative. In other words, this activation function just penalizes the negative inputs. ReLU is the most used activation function in the hidden layers of modern ANNs.
- **Leaky ReLU (Rectified Linear Unit)** activation function: It is a modification of ReLU, where instead of outputting 0 for negative values, Leaky ReLU has a small negative slope. Some networks use Leaky ReLU instead of ReLU.

There are also several other modifications of ReLU activations such as GeLU (Gaussian error Linear Unit) and SeLU (Scaled exponential Linear Unit). Overall, ReLU is still the most common activation function.

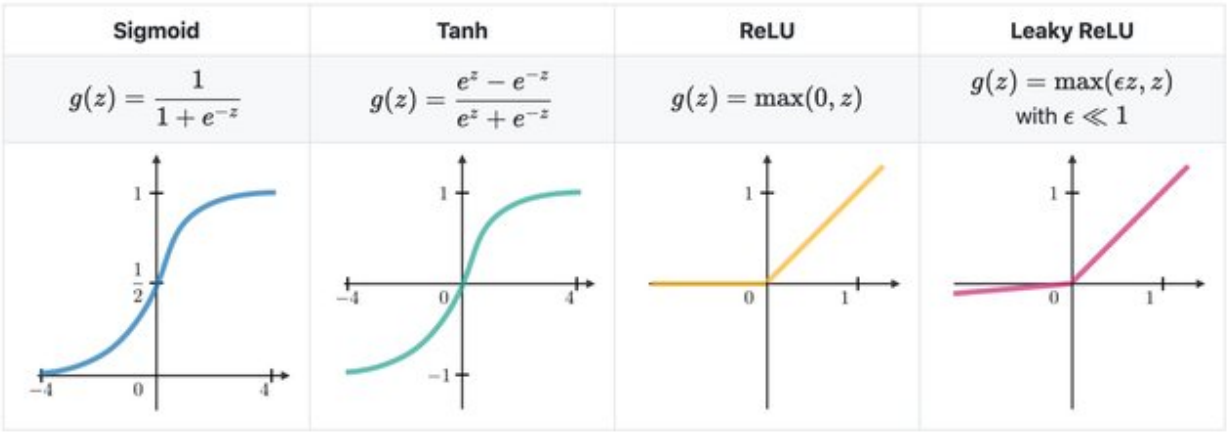


Figure: Activation functions in ANNs. Figure source: link

15.1.3 Training ANNs

ANNs are trained by iteratively updating the network parameters (weights and biases) to minimize the difference between the network predictions and target labels.

Each iteration in the training phase includes the following 4 steps:

1. Forward pass (forward propagation)
2. Loss calculation
3. Error backpropagation (backward pass)
4. Model parameters update

Forward propagation is also known as **forward pass**, because the input data is passed through all hidden layers of the neural network toward the output layer to obtain the network predictions. For instance, if the input data is an image, the image is transformed through the layers of the network, and for classification problems, the output is a vector of predicted class probabilities.

Loss calculation is the second step, in which the loss of the network is calculated as a difference between the network output (predictions for each class) and the target ground-truth label for an image image. Common loss function for classification tasks is *crossentropy loss*, calculated as $-\sum y \cdot log(\hat{y})$, where y is the ground-truth label and $\hat{y}$ is the network prediction. For regression tasks commonly used loss functions include *mean-squared error* (calculated as $\sum (y - \hat{y})^2$) and *mean-absolution error* (calculated as $\sum |y - \hat{y}|$) between the network predictions and target ground-truth values.

Error backpropagation is also called **backward pass** or **backward propagation** (where the term backpropagation is short for backward propagation). In this step, the predicted outputs are traversed back through the network, from the last (output) layer backward toward the first (input) layer. During the backward step, the gradients of the loss with respect to the model parameters $\nabla\mathcal{L}()$ are calculated. If you recall from calculus, **gradient** is the vector of partial derivatives of a function, or for a loss function $\mathcal{L}$ and network parameters , the gradient is the vector $\nabla\mathcal{L} = [\partial\mathcal{L}/\partial_i]$. The gradients quantify the impact of changing the parameters in the network to the predicted outputs. Automatic calculation of the gradients (automatic differentiation) is available in all current deep learning libraries, which significantly simplifies the implementation of deep learning algorithms.

Model parameters update is the last step in which new values for the model parameters are calculated and updated, typically using the **Gradient Descent** algorithm.

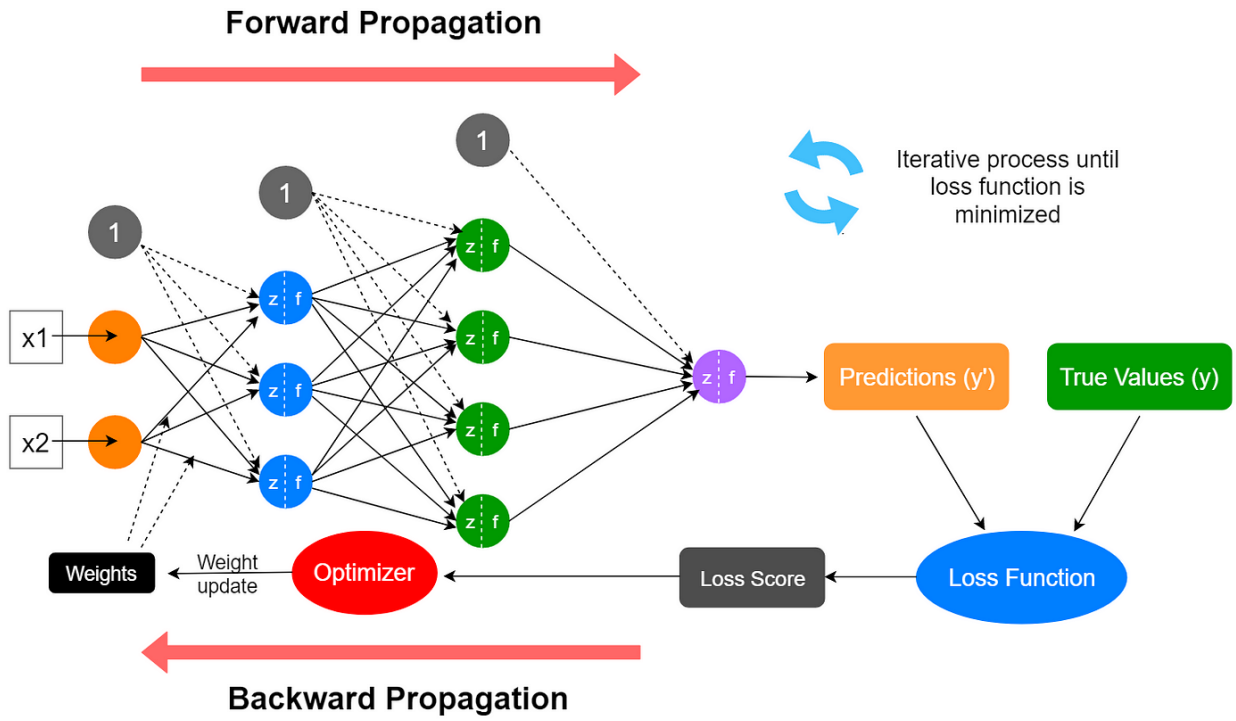
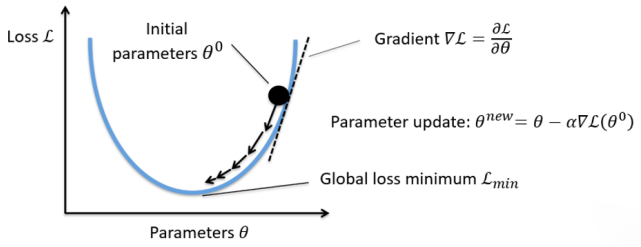


Figure: Steps in training ANNs.

A simple depiction of the **Gradient Descent** algorithm is shown in the next figure. The blue curve is a loss function $\mathcal{L}$, and the gradient of the loss function $\nabla\mathcal{L}$ gives the slope of the function. By updating the parameters in the opposite direction of the gradient of the loss $\nabla\mathcal{L}$, the algorithm finds parameters for which the loss $\mathcal{L}$ has a minimal value.



Gradient descent algorithm:

1. Randomly initialize the parameters θ^0
2. Compute the gradient of the loss function: $\nabla\mathcal{L}(\theta^0)$
3. Update the parameters: $\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$
   - α is learning rate
4. Go to step 2 and repeat

Figure: Steps in training ANNs.

Almost all modern neural networks are trained by applying a modified version of the Gradient Descent algorithm. Examples of such advanced Gradient Descent algorithms include Adam, SGD (Stochastic Gradient Descent), RMSprop, Adagrad, Nadam, and others. The deep learning libraries typically refer to the used algorithms for minimizing the loss of the model as **optimizers**.

Unlike the model training phase, predicting on test data (also known as **inference**) requires only a forward pass through the model, where for a given input instance the model outputs the prediction. For instance, in the above figure the output is a vector of class probabilities.

15.1.4 Types of ANN Architectures

The main types of ANN architectures are:

- **Fully-connected neural networks (FCNNs)** (a.k.a. densely-connected neural networks, or multi-layer perceptrons)
- **Convolutional neural networks (CNNs)**
- **Recurrent neural networks (RNNs)**
- **Transformer neural networks (TNNs)**
- **Graph neural networks (GNNs)**

In this lecture, we will study Fully-connected NNs, and in the following lecture we will explore the other ANN architectures.

15.1.5 Fully-connected NNs

Fully-connected NNs are made of layers in which the neurons of any layer are connected to all other neurons of the preceding and succeeding layer. They are also called **Densely-connected NNs**, as well they are often referred to as **Multi-Layer Perceptrons (MLPs)**.

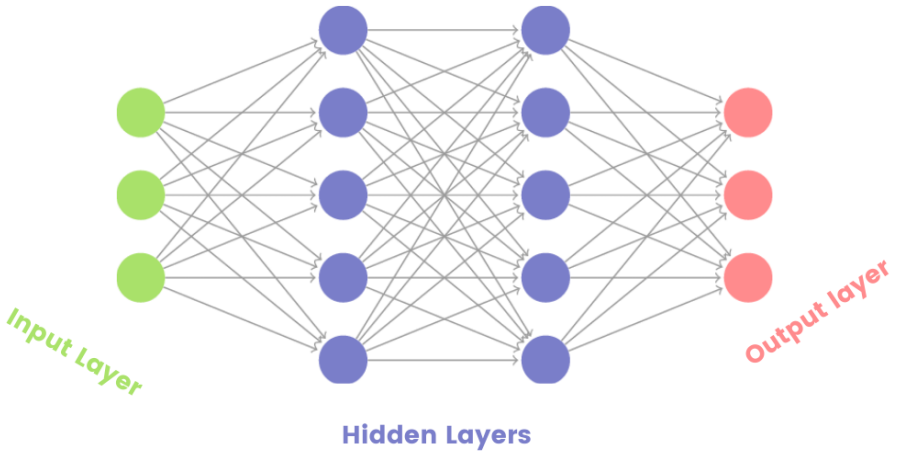


Figure: Fully-connected NNs.

Fully-connected NNs are mostly used in simple tasks, such as for classification or regression tasks with tabular data. Their performance for tasks with images, text data, or other data formats is often inferior in comparison to CNNs, RNNs, TNNs, and GNNs.

7.15.2 15.2 Classification with ANNs

In calssification tasks, the goal is to predict the category, i.e., class label of the data samples.

Classification tasks can be divided into two main types:

- **Binary classification** is classification with two classes. An example is to classify a tweet as positive or negative depending on its content. In the output layer, we need to use a single output neuron with a `sigmoid` activation function to output a value between 0 and 1. A threshold value (by default set to 0.5) can be used to differentiate between positive and negative classes. For example, if the value of the output neuron is 0.7, the tweet can be assigned as positive (since the value is greater than 0.5). The common loss function in binary classification is `binary crossentropy`.
- **Multiclass classification**: is classification with three or more classes. An example is to classify a tweet into several classes, such as news, tech, sports, or weather. In the output layer, we need to use a number of output neurons equal to the number of classes, with a `softmax` activation function. E.g., for the above tweet example, we will use 4 output neurons. Consequently, the output of the network will be a vector of class probabilities whose dimension is equivalent to the number of classes, such as [0.2 0.5 0.1 0.2] for the tweet example. The category with the maximum value is assigned as a class label to the tweet. In this case, if the four categories were news, tech, sports, or weather, the tweet will be assigned the class tech, since the network output of 0.5 for this class is the greatest. The commonly used loss functions in multiclass classification are `categorical crossentropy` (when the target labels are in one-hot matrix encoding format) and `sparse categorical crossentropy` (when the target labels are in ordinal encoding format).

In both binary and multiclass classification problems, the hyperparameters of the ANN related to the number of input neurons, activation functions in the hidden layers, number of hidden layers, etc. depend on the specific problem we are solving.

Table: Typical values of hyperparameters in ANNs for classification

Hyperparameter	Binary classifier	Multiclass classifier
Neurons in input layer	Number of input features	Number of input features
Number of hidden layer(s)	Depends on the problem, typically from 1-10	Depends on the problem, typically from 1-10
Neurons per hidden layer	Depends on the problem, typically 10-100	Depends on the problem, typically 10-100
Neurons in output layer	1	Neurons equivalent to number of classes
Activation in hidden layers	Mostly ReLU	Mostly ReLU
Activation in output layer	sigmoid	softmax
Loss function	binary cross entropy	categorical crossentropy / sparse categorical crossentropy
Optimizer	Mostly Adam, SGD, RMSProp	Mostly: Adam, SGD, RMSProp

ANNs have many hyperparameters, and selecting the best values of the hyperparameters can be challenging. In the next lectures we will explore best practices for hyperparameter finetuning in ANNs.

15.2.1 Binary Classification

We will first implement an ANN for binary classification.

For this purpose, we will use again the Breast Cancer dataset which we also used in the previous lecture. Recall that the dataset has 569 samples, each having 30 features. The target labels are the benign or malignant class, hence it is a binary classification problem.

[1]:
```
# Import libraries
import numpy as np
import pandas as pd
import sklearn
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

import tensorflow as tf
from tensorflow import keras

# Print the version of tensorflow
print("TensorFlow version:{}".format(tf.__version__))
```

```
TensorFlow version:2.14.0
```

Loading the Dataset

The Breast Cancer dataset can be directly loaded from scikit-learn datasets. Let's create a DataFrame `df` having the features and the target column as the last column to the right.

[2]:
```
# Load the dataset
from sklearn.datasets import load_breast_cancer

bc = load_breast_cancer()
df = pd.DataFrame(data=bc.data, columns=bc.feature_names)
df["target"] = bc.target
```

[3]:
```
# Check the shape of the dataset
df.shape
```

[3]:
```
(569, 31)
```

[4]:
```
# Show the first few rows
df.head()
```

[4]:
```
   mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0        17.99         10.38          122.80     1001.0          0.11840
1        20.57         17.77          132.90     1326.0          0.08474
2        19.69         21.25          130.00     1203.0          0.10960
3        11.42         20.38           77.58      386.1          0.14250
4        20.29         14.34          135.10     1297.0          0.10030

   mean compactness  mean concavity  mean concave points  mean symmetry  \
0           0.27760          0.3001              0.14710         0.2419
```

(continues on next page)

(continued from previous page)

```
1           0.07864          0.0869             0.07017         0.1812
2           0.15990          0.1974             0.12790         0.2069
3           0.28390          0.2414             0.10520         0.2597
4           0.13280          0.1980             0.10430         0.1809

   mean fractal dimension  ...  worst texture  worst perimeter  worst area  \
0                 0.07871  ...          17.33           184.60      2019.0
1                 0.05667  ...          23.41           158.80      1956.0
2                 0.05999  ...          25.53           152.50      1709.0
3                 0.09744  ...          26.50            98.87       567.7
4                 0.05883  ...          16.67           152.20      1575.0

   worst smoothness  worst compactness  worst concavity  worst concave points  \
0            0.1622             0.6656           0.7119                0.2654
1            0.1238             0.1866           0.2416                0.1860
2            0.1444             0.4245           0.4504                0.2430
3            0.2098             0.8663           0.6869                0.2575
4            0.1374             0.2050           0.4000                0.1625

   worst symmetry  worst fractal dimension  target
0          0.4601                  0.11890       0
1          0.2750                  0.08902       0
2          0.3613                  0.08758       0
3          0.6638                  0.17300       0
4          0.2364                  0.07678       0

[5 rows x 31 columns]
```

Next, we will extract the features and the target labels, and we will split them into training and testing sets.

[5]:
```
# Extract the data and the labels
X = df.drop('target', axis=1)
y = df['target']
```

[6]:
```
from sklearn.model_selection import train_test_split

train_data, test_data, train_labels, test_labels = train_test_split(X, y, test_size=0.2,␣
↪random_state=42, stratify=y)
```

[7]:
```
print('Training data inputs', train_data.shape)
print('Training labels', train_labels.shape)
print('Testing data inputs', test_data.shape)
print('Testing labels', test_labels.shape)
```

```
Training data inputs (455, 30)
Training labels (455,)
Testing data inputs (114, 30)
Testing labels (114,)
```

We will also scale the data with the `MinMaxScaler`. Recall again that we do not fit the scaler on the test dataset, but we only transform it.

[8]:
```
# Scaling the features to be between 0 and 1.
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

train_data = scaler.fit_transform(train_data)
test_data = scaler.transform(test_data)
```

Create the ANN

To create ANNs in this lectue we will use the Tensorflow-Keras library. Keras is part of TensorFlow, and if you notice in the above cell Keras is imported from TensorFlow.

To define the network architecture in Keras, we will use the data structures **layers** and **models**.

In the cell below we imported the needed layers and model classes and afterward we will create layers and model objects as instances of these classes.

For the ANN for this case we will use `Input` and `Dense` layers, as explained below. `Model` is a general class in Keras that is used to create new ANN models.

[9]:
```
# Import the layers and the model
from keras.layers import Input
from keras.layers import Dense
from keras.models import Model
```

TensorFlow-Keras requires the first layer in an ANN to be an **Input** layer, which will provide the shape of the input data in the form of a tuple. For the Breast Cancer dataset, there are 30 input features for training the model, and therefore, we will set the shape of the input data to `(30,)`.

Note also that to create and train an ANN, we don't need to provide information about the number of samples that are available for training or testing the model.

Let's add two hidden layers. In Keras, `Dense` layers are fully-connected layers. Therefore, the layers `dense1` and `dense2` below are fully-connected layers, of which, the first layer has 30 units (or neurons), and the second layer has 15 neurons. The second argument in the Dense layers `activation` defines the activation function that is applied at each layer, and in this case we selected ReLU activations as the most common activation function. For each layer in Keras, we need to also define the previous layer to which it connects. In this case, for `dense1` the previous layer is the `inputs` layer, and `dense2` follows the `dense1` layer.

Finally, we added an output layer which is also a Dense layer. As we mentioned, for binary classification problems, the output layer has 1 neuron and it uses a `sigmoid` activation function. Preceding layer to the `outputs` layer is `dense2` layer.

[10]:
```
# Define the layers in the network
inputs = Input(shape=(30,))
dense1 = Dense(units=30, activation='relu')(inputs)
dense2 = Dense(units=15, activation='relu')(dense1)
outputs = Dense(units=1, activation='sigmoid')(dense2)
```

After we define the layers, we will create a new model as an instance of the `Model` class. Let's name the new instance `model_1`. The arguments in `Model` are the input and output layers, which we named in the above cell `inputs` and `outputs`.

[11]:
```
# Define the model by providing the inputs and outputs
model_1 = Model(inputs, outputs)
```

Now that we have created a model, we can inspect its architecture with the `.summary()` method. It provides a table with all layers in the network, the shape of the tensors that are output by each layer, and the number of parameters in each layer.

We can see that the first layer in the model expects the passed data to be of shape `(None, 30)`. Here `None` is a placeholder, and specifies that the model can accept any number of samples that have 30 elements.

The output of the model is of shape `(None, 1)`, meaning that it will output one single value for each data sample.

Also, we can see in the information under the table that this model has 1,411 trainable parameters, of which 930 are in the first hidden layer, 465 are in the second hidden layer, and 16 are in the output layer. During the model training, we will try to find optimal values of the network parameters that minimize the misclassification of the samples in the training dataset.

[12]:
```
# Model summary
model_1.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 30)]              0

 dense (Dense)               (None, 30)                930

 dense_1 (Dense)             (None, 15)                465

 dense_2 (Dense)             (None, 1)                 16

=================================================================
Total params: 1411 (5.51 KB)
Trainable params: 1411 (5.51 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

The graph of the network is also shown in the following figure. We can see the layers with all 1,411 connections in the network.

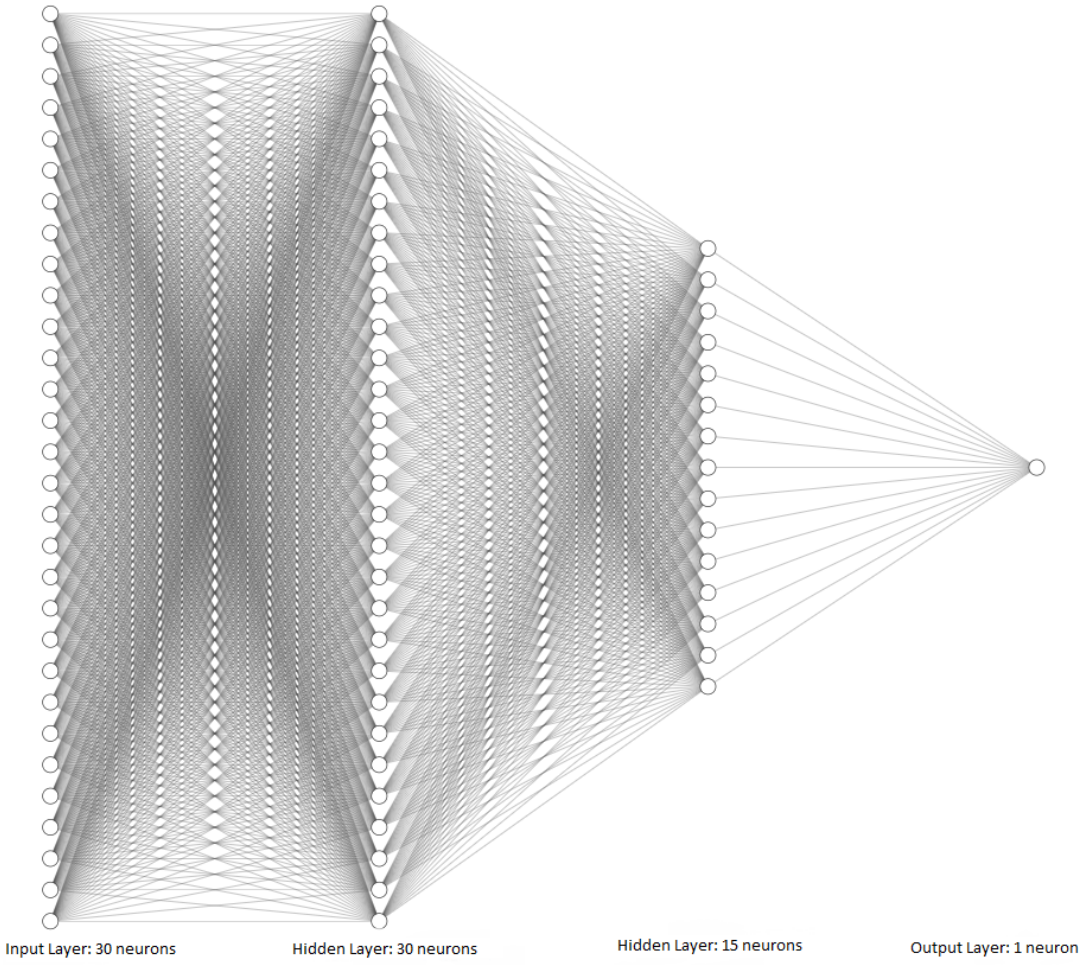


Figure: Graphical depiction of the network.

Compile the ANN

To be able to train the ANN model, first we need to compile it. During the compiling step, we will specify the loss function, optimizer, and metric.

The **loss function** (cost function) calculates the difference between the predicted class label by the model and the ground-truth target class label for each sample. If the difference is high, this means a higher loss.

In Keras and TensorFlow, there are three main crossentropy losses:

- `binary_crosssentropy` used for binary classification with two classes.
- `categorical_crossentropy` used for multiclass classification with three and more classes, and the target labels are encoded in one-hot matrix format.
- `sparse_categorical_crossentropy` used for multiclass classification with three and more classes, and the target labels are encoded in ordinal format.

The **optimizer** is the optimization algorithm that is used to reduce the loss function. As we explained earlier, several different optimization algorithms are used with ANNs. In this case we will use `Adam` optimizer (which stands for Adaptive Moment Estimation). It is one of the fastest and most commonly used optimizers at present.

We will use accuracy as a **metric**, but we can also use AUC-ROC, or F1 score, or other metrics.

[13]:
```
model_1.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
```

Train the ANN

To train the model, Keras uses the `fit` method as in scikit-learn. Required arguments in `fit` include the training dataset and training labels, and additional arguments can include the number of epochs, batch size, verbose, and others.

The **batch size** below means that the `fit` method will repeatedly randomly select 64 samples from the training dataset, and use the batch of samples to train the model and update the model parameters. Therefore, for the training dataset of 455 samples, the updates will be repeated 455/64 = 7.1 times (i.e., the model will be updated 8 times to process the entire dataset). When all 455 samples in the training dataset are used, that is considered one **epoch**. In this case, the training will continue for 100 epochs. Therefore, the parameters in the model will be updated 800 times in total (100 epoch x 8 steps per epoch = 800). Or, we can say that there are 800 training steps.

The `verbose` argument is set to 0, and the results from the training will not be displayed after the cell. Setting `verbose` to 1 will display a progress bar, and the default value of 2 will display the training loss and accuracy for each epoch.

Also, in the next cell we assign the output of the `fit` method to `history`, that is, we will save the training results in this variable, and later we will use it to plot the learning curves and analyze the model training.

[14]:
```
history = model_1.fit(train_data, train_labels, epochs=100, batch_size=64, verbose=0)
```

Evaluate the ANN on Test Data

Keras has a method `evaluate` that we can use for calculating the accuracy of the model. The arguments are `test_data` and `test_labels`. Note that `evaluate()` returns two values: the loss and accuracy of the model. Therefore, to print the accuracy we used the element with index 1 in `evals_test`.

[15]:
```
# Evaluate on test dataset
evals_test = model_1.evaluate(test_data, test_labels)
print("Classification Accuracy: ", evals_test[1])
```

```
4/4 [==============================] - 0s 3ms/step - loss: 0.0708 - accuracy: 0.9649
Classification Accuracy:  0.9649122953414917
```

[16]:
```
# evals_test contains the loss and accuracy of the model
evals_test
```

[16]:
```
[0.0707874745130539, 0.9649122953414917]
```

To obtain the predictions by the model on data samples, we will use the `predict` method in Keras, which works similarly to the `predict` method in scikit-learn.

[17]:
```
predictions = model_1.predict(test_data)
```

```
4/4 [==============================] - 0s 0s/step
```

Let's check the shape of `predictions` to understand the output of the model, and compare it to the shape of `test_data` that we used in `predict()`. The test dataset has 114 samples, each with 30 features. Therefore, for each sample the model made a prediction and outputted one single value.

[18]:
```
predictions.shape
```

[18]:
```
(114, 1)
```

[19]:
```
# Compare to the shape of test_data
test_data.shape
```

[19]:
```
(114, 30)
```

Let's inspect the output for the first 6 samples in `test_data` in the next cell. The shown values are difficult to interpret because they are displayed in scientific notation, therefore, in the following cell the values are shown rounded to 3 decimal places.

[20]:
```
predictions[:6]
```

[20]:
```
array([[1.8629966e-04],
       [9.9987435e-01],
       [7.2749399e-02],
       [7.1462798e-01],
       [1.4644288e-04],
       [9.5147675e-01]], dtype=float32)
```

[21]:
```
np.around(predictions[:6], 3)
```

[21]:
```
array([[0.   ],
       [1.   ],
       [0.073],
       [0.715],
       [0.   ],
       [0.951]], dtype=float32)
```

Now we can notice that the predictions by the model have values between 0 and 1, and they can be considered as the probability of each sample to be either benign (0) or malignant (1).

This form of the outputs is expected, because we used a `sigmoid` activation function in the output layer, which restricts the outputs to be in the range between 0 and 1.

To assign the class label 0 or 1 to the predicted values, we can round the predictions with `np.round()` or `tf.math.round()` to return the closest integer. For example, for the prediction of 0.016, the class label is 0. For the prediction of 0.64, the class label is 1.

[22]:
```
# Class labels for the first  predicted values
np.round(predictions[:6])
```

[22]:
```
array([[0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.]], dtype=float32)
```

We can also calculate the accuracy in the same way as did with scikit-learn models, by using the `accuracy_score` function. As expected, the calculated value for the accuracy is the same as the one calculated in the Keras `evaluate` method.

[23]:
```
from sklearn.metrics import accuracy_score

predictions = model_1.predict(test_data)

accuracy = accuracy_score(test_labels, np.round(predictions))
print('The test accuracy is {0:6.4f} %'.format(accuracy*100))
```

```
4/4 [==============================] - 0s 1ms/step
The test accuracy is 96.4912 %
```

To better understand the model performance, we can also display the confusion matrix.

[24]:
```
# Getting the confusion matrix
from sklearn.metrics import confusion_matrix

confmat = confusion_matrix(test_labels, np.round(predictions), labels=[1,0])
print(confmat)
```

```
[[70  2]
 [ 2 40]]
```

As well as, we can use the function `classification_report` from scikit-learn, to display the precision, recall, F1-score, macro average accuracy, and weighted average accuracy per class.

[25]:
```
# Classification report: F1 score, Recall, Precision
from sklearn.metrics import classification_report

print(classification_report(test_labels, np.round(predictions)))
```

```
              precision    recall  f1-score   support

           0       0.95      0.95      0.95        42
           1       0.97      0.97      0.97        72

    accuracy                           0.96       114
   macro avg       0.96      0.96      0.96       114
weighted avg       0.96      0.96      0.96       114
```

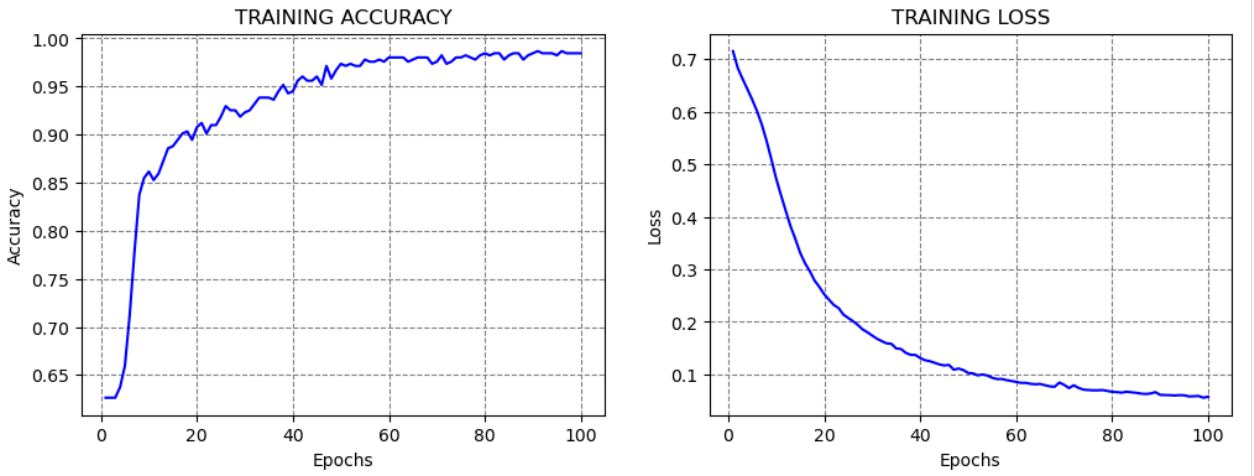
Visualize the Training Loss and Accuracy

The following figure visualizes the history of model training, and shows the training accuracy and loss of the model for each of the 100 training epochs. For this purpose, we used the `history` variable that we assigned to `fit` when training the model. It contains an attribute `history.history` which is a dictionary that stores the loss and accuracy of the model for each epoch.

These plots are called **learning curves** and are important for understanding the model performance. It is a good practice to always create a plot after training a model and analyze the learning curves.

For instance, we can notice that at the end of the 100th epoch, the training loss is still decreasing. Thus, we could have trained the model for longer than 100 epochs, and this could have resulted in higher accuracy. In the next lectures we will describe best practices for determining the number of epochs.

[26]:
```
# plot the accuracy and loss
train_loss = history.history['loss']
acc = history.history['accuracy']

epochsn = np.arange(1, len(train_loss)+1,1)
plt.figure(figsize=(12, 4))

plt.subplot(1,2,1)
plt.plot(epochsn, acc, 'b')
plt.grid(color='gray', linestyle='--')
plt.title('TRAINING ACCURACY')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.subplot(1,2,2)
plt.plot(epochsn,train_loss, 'b')
plt.grid(color='gray', linestyle='--')
plt.title('TRAINING LOSS')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```

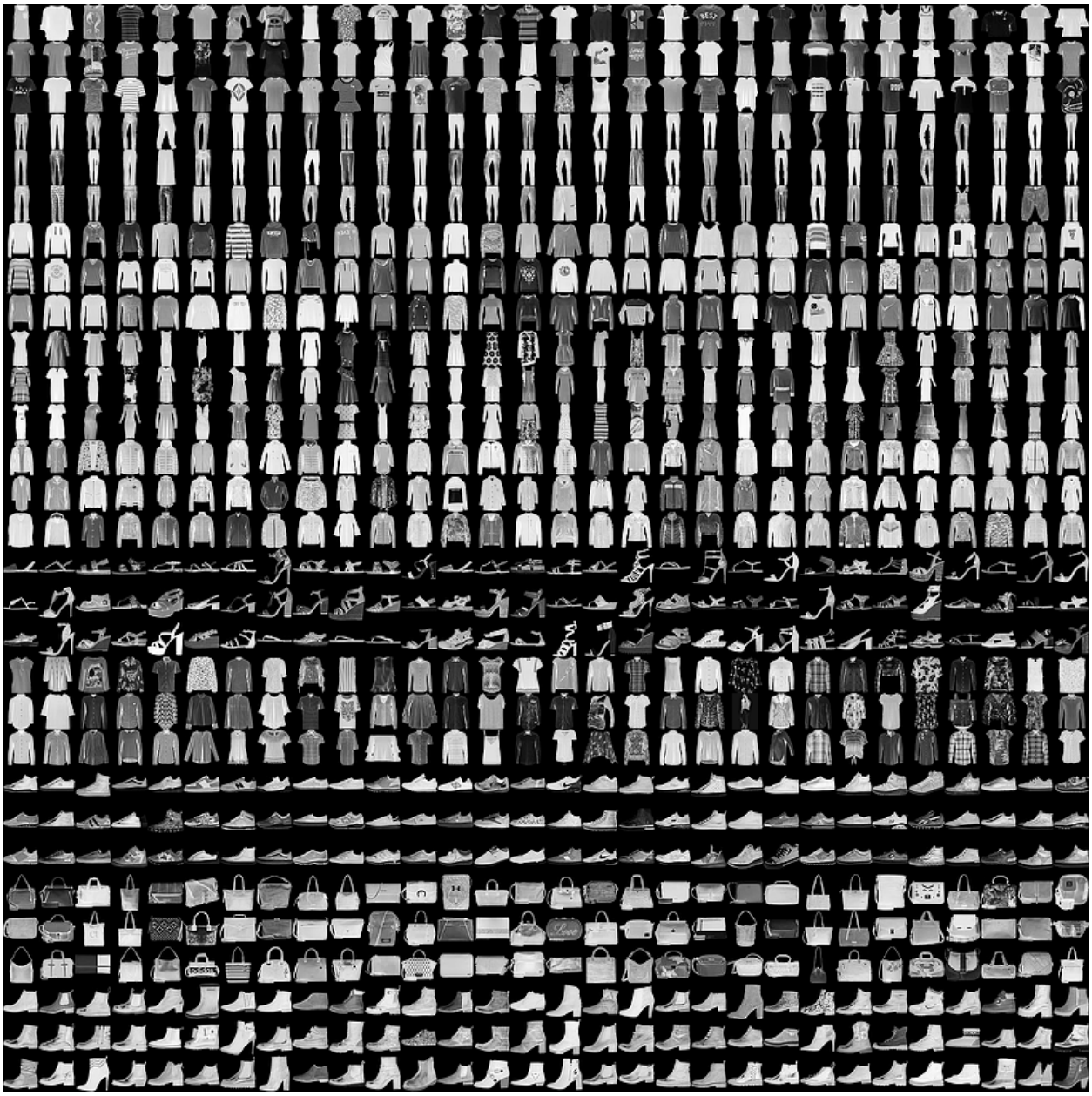


15.2.2 Multiclass Classification

In this section, we will consider a multiclass classification problem. Our goal will be to train an ANN model for classifying the images in the Fashion MNIST dataset into 10 classes.

Loading the Dataset

Fashion MNIST dataset has 70,000 images, of which 60,000 are allocated for the training set and 10,000 for the test set. Examples of images in the dataset are shown in the next figure. Each is a grayscale images with 28 x 28 pixels.



The images are classified into the following 10 categories:

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

The dataset can be imported directly from Keras datasets.

[27]:
```
# Load the Fashion MNIST dataset
fashion_mnist = tf.keras.datasets.fashion_mnist

(fashion_train, fashion_train_label), (fashion_test, fashion_test_label) = fashion_mnist.
↪load_data()
```

[28]:
```
print('Training data inputs', fashion_train.shape)
print('Training labels', fashion_train_label.shape)
print('Testing data inputs', fashion_test.shape)
print('Testing labels', fashion_test_label.shape)
```

```
Training data inputs (60000, 28, 28)
Training labels (60000,)
Testing data inputs (10000, 28, 28)
Testing labels (10000,)
```

Let's check the shape of the training data and the values in the first image.

[29]:
```
# Shape of each sample
fashion_train[0].shape
```

[29]:
```
(28, 28)
```

[30]:
```
# Pixels in the first train image
fashion_train[0]
```

[30]:
```
array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,
          0,   0,  13,  73,   0,   0,   1,   4,   0,   0,   0,   0,   1,
          1,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,
```

(continues on next page)

(continued from previous page)

```
          0,  36, 136, 127,  62,  54,   0,   0,   0,   1,   3,   4,   0,
          0,   3],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   6,
          0, 102, 204, 176, 134, 144, 123,  23,   0,   0,   0,   0,  12,
         10,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0, 155, 236, 207, 178, 107, 156, 161, 109,  64,  23,  77, 130,
         72,  15],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,
         69, 207, 223, 218, 216, 216, 163, 127, 121, 122, 146, 141,  88,
        172,  66],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   1,   1,   0,
        200, 232, 232, 233, 229, 223, 223, 215, 213, 164, 127, 123, 196,
        229,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
        183, 225, 216, 223, 228, 235, 227, 224, 222, 224, 221, 223, 245,
        173,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
        193, 228, 218, 213, 198, 180, 212, 210, 211, 213, 223, 220, 243,
        202,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   3,   0,  12,
        219, 220, 212, 218, 192, 169, 227, 208, 218, 224, 212, 226, 197,
        209,  52],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   6,   0,  99,
        244, 222, 220, 218, 203, 198, 221, 215, 213, 222, 220, 245, 119,
        167,  56],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   4,   0,   0,  55,
        236, 228, 230, 228, 240, 232, 213, 218, 223, 234, 217, 217, 209,
         92,   0],
        [  0,   0,   1,   4,   6,   7,   2,   0,   0,   0,   0,   0, 237,
        226, 217, 223, 222, 219, 222, 221, 216, 223, 229, 215, 218, 255,
         77,   0],
        [  0,   3,   0,   0,   0,   0,   0,   0,   0,  62, 145, 204, 228,
        207, 213, 221, 218, 208, 211, 218, 224, 223, 219, 215, 224, 244,
        159,   0],
        [  0,   0,   0,   0,  18,  44,  82, 107, 189, 228, 220, 222, 217,
        226, 200, 205, 211, 230, 224, 234, 176, 188, 250, 248, 233, 238,
        215,   0],
        [  0,  57, 187, 208, 224, 221, 224, 208, 204, 214, 208, 209, 200,
        159, 245, 193, 206, 223, 255, 255, 221, 234, 221, 211, 220, 232,
        246,   0],
        [  3, 202, 228, 224, 221, 211, 211, 214, 205, 205, 205, 220, 240,
         80, 150, 255, 229, 221, 188, 154, 191, 210, 204, 209, 222, 228,
        225,   0],
        [ 98, 233, 198, 210, 222, 229, 229, 234, 249, 220, 194, 215, 217,
        241,  65,  73, 106, 117, 168, 219, 221, 215, 217, 223, 223, 224,
        229,  29],
        [ 75, 204, 212, 204, 193, 205, 211, 225, 216, 185, 197, 206, 198,
        213, 240, 195, 227, 245, 239, 223, 218, 212, 209, 222, 220, 221,
        230,  67],
        [ 48, 203, 183, 194, 213, 197, 185, 190, 194, 192, 202, 214, 219,
        221, 220, 236, 225, 216, 199, 206, 186, 181, 177, 172, 181, 205,
```

(continues on next page)

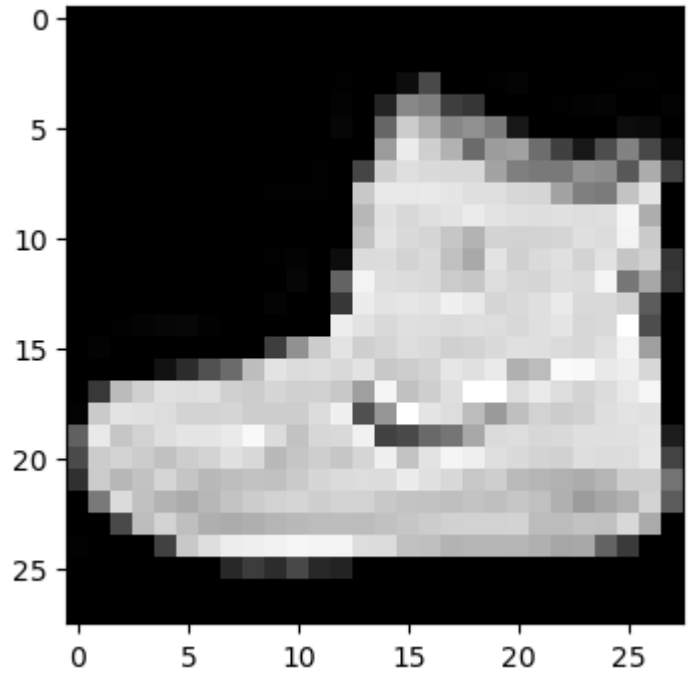
(continued from previous page)

```
        206, 115],
       [  0, 122, 219, 193, 179, 171, 183, 196, 204, 210, 213, 207, 211,
        210, 200, 196, 194, 191, 195, 191, 198, 192, 176, 156, 167, 177,
        210,  92],
       [  0,   0,  74, 189, 212, 191, 175, 172, 175, 181, 185, 188, 189,
        188, 193, 198, 204, 209, 210, 210, 211, 188, 188, 194, 192, 216,
        170,   0],
       [  2,   0,   0,   0,  66, 200, 222, 237, 239, 242, 246, 243, 244,
        221, 220, 193, 191, 179, 182, 182, 181, 176, 166, 168,  99,  58,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,  40,  61,  44,  72,  41,  35,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0]], dtype=uint8)
```

[31]:
```
# A list of label names
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

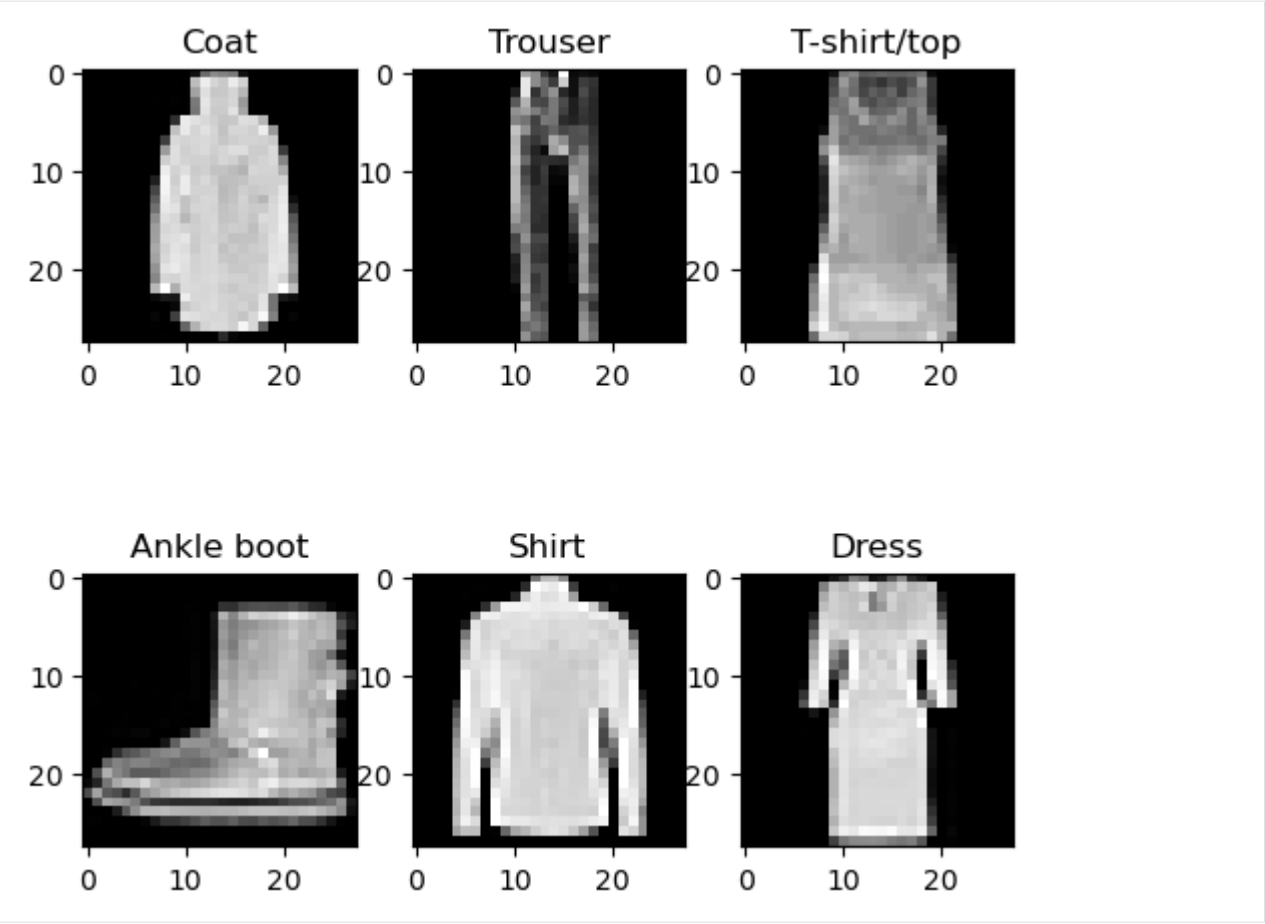
[32]:
```
# Show the first image
index = 0
plt.figure(figsize=(4,4))
plt.imshow(fashion_train[index], cmap='gray')
# Display the label
image_label = fashion_train_label[index]
print('This type of image is: {}({})'.format(class_names[image_label], image_label))
```

```
This type of image is: Ankle boot(9)
```



We can also visualize several random images, to ensure that the labels match the images.

[33]:
```
import random

plt.figure(figsize=(6,6))

for index in range(6):
    ax = plt.subplot(2, 3, index+1)
    random_index = random.choice(range(len(fashion_train)))
    plt.imshow(fashion_train[random_index], cmap='gray')
    plt.title(class_names[fashion_train_label[random_index]])
```



Data Preprocessing

As expected, the pixel values in the images range from 0 to 255. To scale the values to the range from 0 to 1, we can just divide the training and test sets by `255.0`.

[34]:
```
# Scaling the image pixels to be between 0 and 1
fashion_train = fashion_train/255.0
fashion_test = fashion_test/255.0
```

In addition, to create an ANN with fully-connected layers, we will need to reshape the images into one-dimensional arrays. Since each image has 28 x 28 = 784 pixels, let's use `reshape` to flatten the images.

In the next lecture we will study Convolutional Neural Networks which are particularly suitable for processing image data, and they can work with 2-dimensional images without the need to flatten the arrays.

[35]:
```
fashion_train = fashion_train.reshape(-1, 784)
```

[36]:
```
fashion_test = fashion_test.reshape(-1, 784)
```

[37]:
```
# Check the shape
fashion_train.shape
```

[37]:
```
(60000, 784)
```

Create the ANN

In the next cell, we define the layers in the ANN. The architecture of the model is similar to the model we used in the previous example. Note that in the arguments for `Dense` layers we don't need to specify the `units` keyword, as the first positional argument is the number of units (neurons).

Since this is a multiclass classification problem, notice that the number of neurons in output layer is 10, as it needs to match the number of class labels in the dataset. Also, we need to use `softmax` activation in the output layer.

[38]:
```
# Define the layers in the network
inputs = Input(shape=(784,))
dense1 = Dense(64, activation='relu')(inputs)
dense2 = Dense(32, activation='relu')(dense1)
outputs = Dense(10, activation='softmax')(dense2)
```

[39]:
```
fashion_classifier = Model(inputs, outputs)
```

Compile the ANN

To compile the model, we used the loss `sparse_categorical_crossentropy` because the labels are integers in ordinal encoding format. If the labels were in one-hot encoding format, we would have used `categorical_crossentropy` loss.

[40]:
```
# Compile the model
fashion_classifier.compile(optimizer='adam',
                           loss='sparse_categorical_crossentropy',
                           metrics=['accuracy'])
```

Train the ANN

To train the model, we pass the training data and labels. In this case, let's train the model for 10 epochs, and use a `batch_size` of 32 images. Also, we used the default verbose, and the training results are displayed at the end of each epoch.

[41]:
```
history = fashion_classifier.fit(fashion_train, fashion_train_label, epochs=10, batch_
↪size=32)
```

```
Epoch 1/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5234 - accuracy: 0.
↪8152
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3838 - accuracy: 0.
↪8618
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3481 - accuracy: 0.
↪8727
Epoch 4/10
```

(continues on next page)

(continued from previous page)

```
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3262 - accuracy: 0.
↪8802
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3093 - accuracy: 0.
↪8869
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2945 - accuracy: 0.
↪8914
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2857 - accuracy: 0.
↪8942
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2746 - accuracy: 0.
↪8977
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2669 - accuracy: 0.
↪9016
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2586 - accuracy: 0.
↪9028
```

Evaluate the ANN on Test Dataset

By using the `evaluate` method in Keras, we can see that the classification accuracy on test data is 88.4%.

[42]:
```
# Evaluate on test dataset
evals_test = fashion_classifier.evaluate(fashion_test, fashion_test_label)
print("Classification Accuracy: ", evals_test[1])
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.3391 - accuracy: 0.8843
Classification Accuracy:  0.8842999935150146
```

We can also use `predict()` to obtain the model's predictions.

[43]:
```
predictions = fashion_classifier.predict(fashion_test)
```

```
313/313 [==============================] - 0s 1ms/step
```

In the next call, we checked the shape of the predicted outputs, and since there are 10,000 test images and 10 classes, the output is `(10000, 10)`.

[44]:
```
# check the shape of the predictions
predictions.shape
```

[44]:
```
(10000, 10)
```

Also, we displayed the predictions for the first 5 test images in the next cell. For each image, the model outputs probabilities for each of the 10 classes. The probabilities sum to 1. For instance, for the first test image, the model assigned the highest probability to the class with index 9.

[45]:
```
# Display the predictions for the first 5 test images
np.around(predictions[:5],3)
```

[45]:
```
array([[0.   , 0.   , 0.   , 0.   , 0.   , 0.015, 0.   , 0.031, 0.   ,
        0.954],
       [0.   , 0.   , 0.996, 0.   , 0.002, 0.   , 0.002, 0.   , 0.   ,
        0.   ],
       [0.   , 1.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   ],
       [0.   , 1.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ,
        0.   ],
       [0.087, 0.   , 0.002, 0.   , 0.   , 0.   , 0.91 , 0.   , 0.   ,
        0.   ]], dtype=float32)
```

To output the indices with the highest probability for each image we can use `np.argmax`.

[46]:
```
# Display the index with the highest probability for the first 5 test images
np.argmax(predictions[:5], axis=1)
```

[46]:
```
array([9, 2, 1, 1, 6], dtype=int64)
```

The ground-truth labels are also shown in the next cell, and they match correctly the predictions for the first 5 test images.

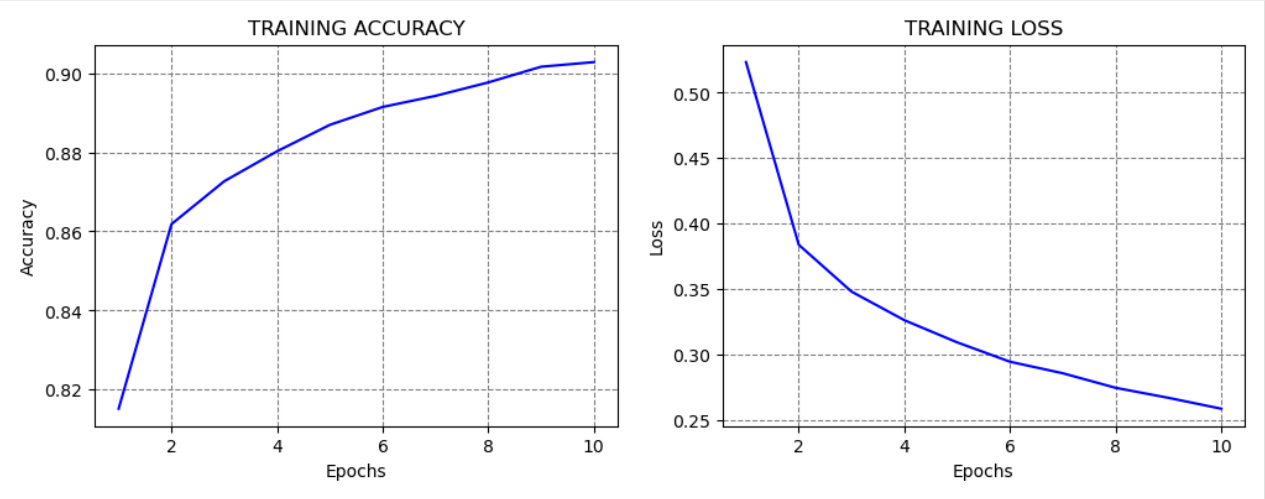
[47]:
```
# print the ground-truth label for the first 5 test images
fashion_test_label[:5]
```

[47]:
```
array([9, 2, 1, 1, 6], dtype=uint8)
```

Visualize the Training Loss and Accuracy

The learning curves are shown in the figure below. It becomes obvious that if we train the model for longer than 10 epochs, we can improve the classification performance.

[48]:
```
# plot the accuracy and loss
train_loss = history.history['loss']
acc = history.history['accuracy']

epochsn = np.arange(1, len(train_loss)+1,1)
plt.figure(figsize=(12, 4))

plt.subplot(1,2,1)
plt.plot(epochsn, acc, 'b')
plt.grid(color='gray', linestyle='--')
plt.title('TRAINING ACCURACY')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.subplot(1,2,2)
plt.plot(epochsn,train_loss, 'b')
plt.grid(color='gray', linestyle='--')
plt.title('TRAINING LOSS')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```



Re-train and Evaluate the ANN

Note that if we re-train the same model on the same training data and evaluate it on the same testing data, in most cases we will obtain similar, but different values for the accuracy!!!

The reason for this is because every time a model is trained, the values of the network parameters are randomly initialized to small values close to 0 (but not 0 though). Because of the random initialization, the model will converge to a different solution every time we train it.

Consequently, ANNs have a stochastic nature, meaning that every time we train a model we obtain a different output.

If the repeatability of the results is important, we can use a random seed to initialize the model parameters, and in that case, the accuracy will be the same if we re-train the model.

[49]:
```
fashion_classifier_2 = Model(inputs, outputs)

# Compile the model
fashion_classifier_2.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

# Train the model
fashion_classifier_2.fit(fashion_train, fashion_train_label, epochs=10, batch_size=32,␣
↪verbose=0)

# Evaluate on test dataset
evals_test = fashion_classifier_2.evaluate(fashion_test, fashion_test_label)
print("Classification Accuracy: ", evals_test[1])
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.3802 - accuracy: 0.8761
Classification Accuracy:  0.8761000037193298
```

7.15.3 15.3 Regression with ANNs

In regression tasks, we are interested in predicting single or multiple continuous values. The ANN architectures for regression tasks are similar to the architectures for classification tasks. The listed recommendations for the input and hidden layers are the same as for classification tasks. The number of neurons in the output layer depends on the problem. If we are predicting a single value it will be 1, and if we are predicting multiple values, the number of output neurons will be set to the number of predicted values.

The choice of activation functions depends on the problem, but in most cases, `ReLU` will work well in the hidden layers. Unlike ANNs for classification, ANNs for regression don't need to have an activation function in the output layer, since the ANN outputs are continuous value(s).

We stated earlier that the loss function used in regression is usually `Mean Squared Error(MSE)`, or if the dataset contains outliers, `Mean Absolute Error (MAE)` loss may be preferred.

A suitable choice for an optimizer is Adam. Other optimizers to try include SGD, RMSProp, Nadam, and others.

Table: Typical values of hyperparameters in ANNs for regression

Hyperparameter	Typical value
Neurons in input layer	Same as the number of input features
Number of hidden layer(s)	Depends on the problem, typically from 1 to 10
Neurons per hidden layer	Depends on the problem, typically from 10 to 100
Activation in hidden layers	Mostly ReLU
Activation in output layer	None in most cases
Loss function	MSE or MAE
Optimizer	Mostly Adam, SGD, RMSProp

Next, we will present an example of regression with ANNs, and for this purpose, we will use the California Housing Dataset that is available in scikit-learn.

We will load the California Housing Dataset from scikit-learn as a pandas DataFrame named `housing`. The objective of the regression task is to predict the median house value, shown in the column to the right.

[50]:
```
from sklearn.datasets import fetch_california_housing

housing = fetch_california_housing(as_frame=True).frame
housing.head()
```

[50]:
```
   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88
1  8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86
2  7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85
3  5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85
4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85

   Longitude  MedHouseVal
0    -122.23        4.526
1    -122.22        3.585
2    -122.24        3.521
3    -122.25        3.413
4    -122.25        3.422
```

[51]:
```
# Check the shape of the dataset
housing.shape
```

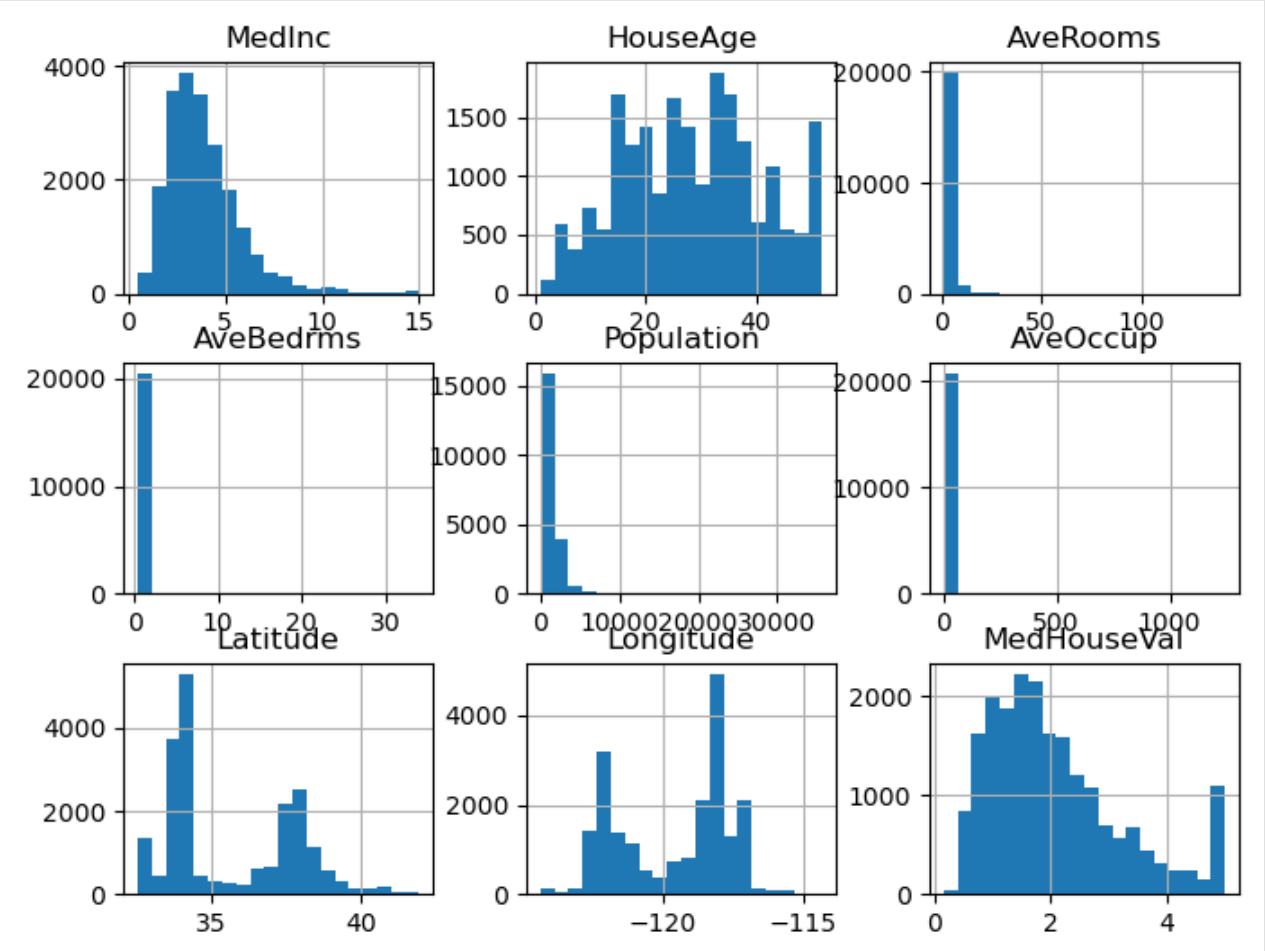
[51]:
```
(20640, 9)
```

We can check if there are missing values, and visualize the histograms of the features.

[52]:
```
housing.isnull().sum()
```

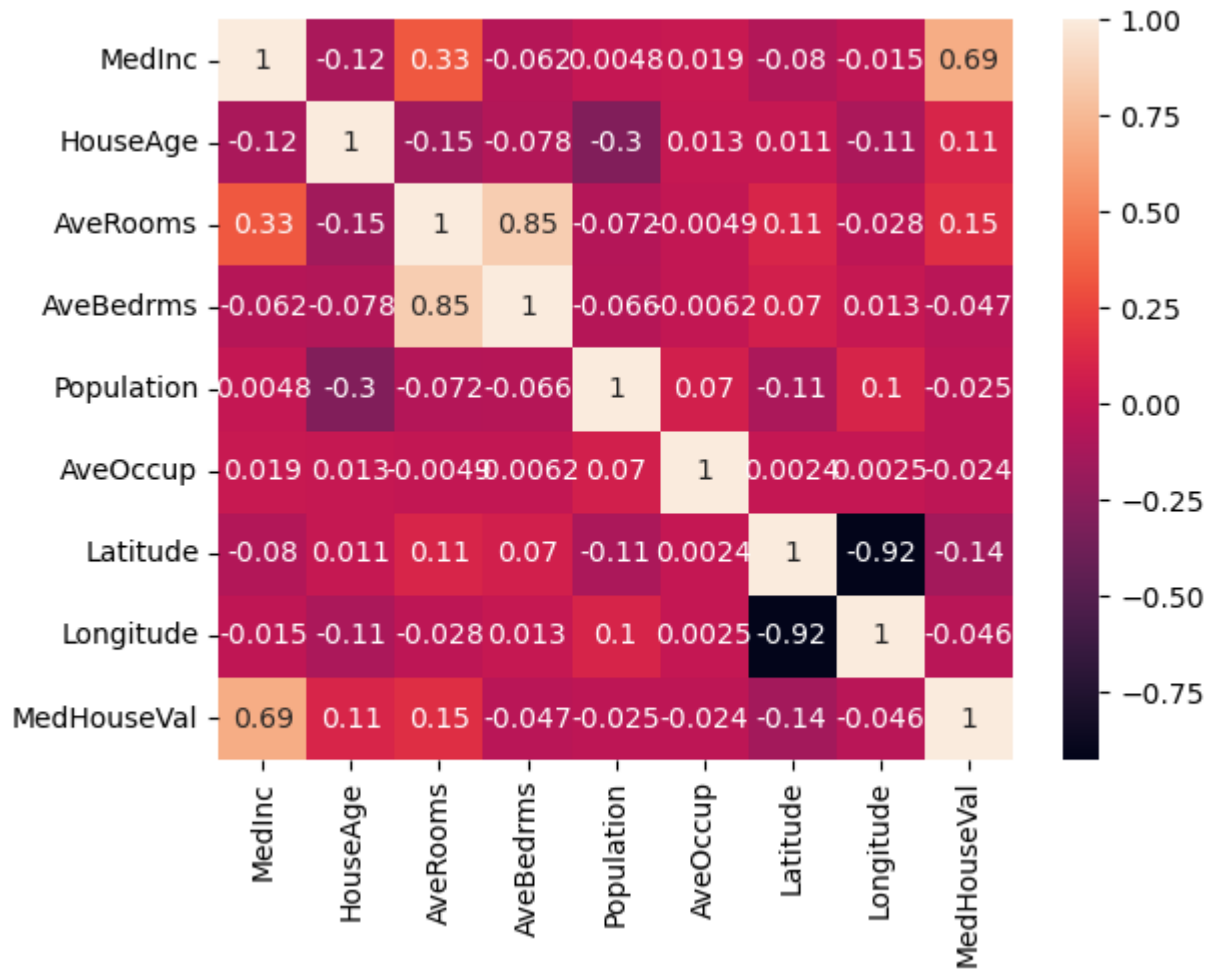
[52]:
```
MedInc          0
HouseAge        0
AveRooms        0
AveBedrms       0
Population      0
AveOccup        0
Latitude        0
Longitude       0
MedHouseVal     0
dtype: int64
```

[53]:
```
housing.hist(bins=20, figsize=(8,6))
plt.show()
```

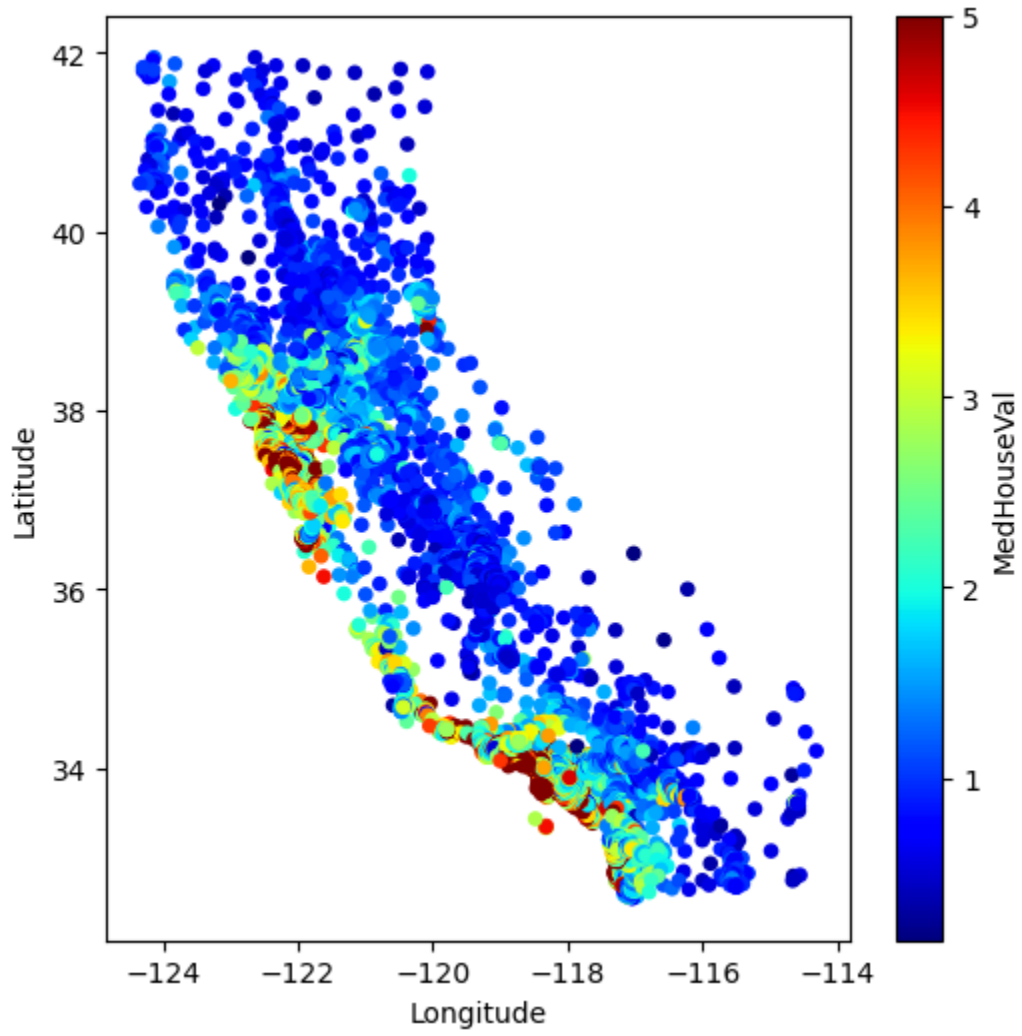


In the next cell we will check if there are correlated features. We can notice high correlation between the house value and the number of bedrooms, which is expected. We can also notice high negative correlation between the price and longitude and latitude. In the following cell we plotted the house value versus longitude and latitude. The plot looks like the map of California, and we can see that houses that are closer to the coast are more expensive.

[54]:
```
correlation = housing.corr()
sns.heatmap(correlation, annot=True);
# plt.show()
```

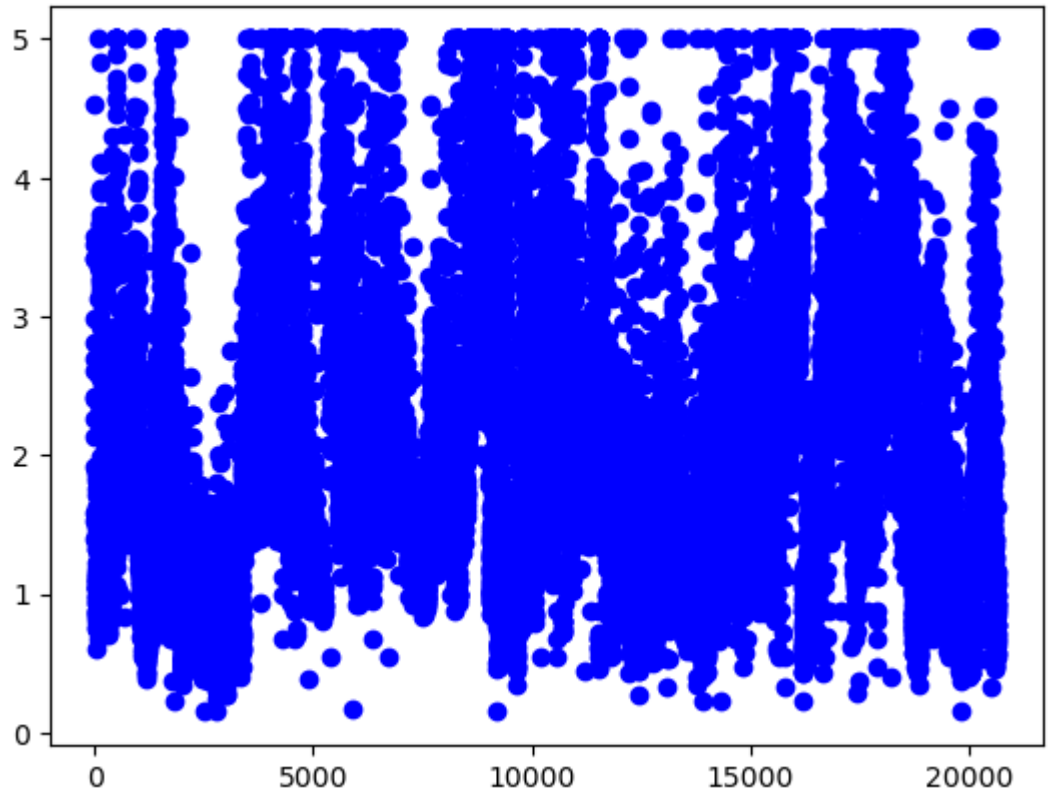


[55]:
```
housing.plot(kind="scatter", x="Longitude", y="Latitude", c="MedHouseVal", cmap="jet",␣
↪colorbar=True, legend=True, sharex=False, figsize=(6,6),)
plt.show()
```



And, if we check the median house value column, we can see that the values are in the range from 0 to 5.

[56]:
```
plt.plot(housing['MedHouseVal'], 'bo')
plt.show()
```



Let's get the features and the target values, and create training and testing datasets.

[57]:
```
X = housing.drop('MedHouseVal', axis=1)
y = housing['MedHouseVal']
```

[58]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

[59]:
```
print('Training data inputs', X_train.shape)
print('Training labels', y_train.shape)
print('Testing data inputs', X_test.shape)
print('Testing labels', y_test.shape)
```

```
Training data inputs (16512, 8)
Training labels (16512,)
Testing data inputs (4128, 8)
Testing labels (4128,)
```

[60]:
```
# Scaling the features to be between 0 and 1
scaler = MinMaxScaler()

X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Let's use a similar ANN architecture as in the classification tasks, with two Dense layers with ReLU activations, having 32 neurons each.

Since we would like to predict a single value, we will use 1 neuron in the output layer, without activation function.

[61]:
```
# Define the layers in the network
inputs = Input(shape=(8,))
dense1 = Dense(32, activation='relu')(inputs)
dense2 = Dense(32, activation='relu')(dense1)
outputs = Dense(1)(dense2)
```

[62]:
```
# Define the model by providing the inputs and outputs
model_2 = Model(inputs, outputs)
```

We can inspect the architecture of the ANN with the `summary()` method.

[63]:
```
model_2.summary()
```

```
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 8)]               0

 dense_6 (Dense)             (None, 32)                288

 dense_7 (Dense)             (None, 32)                1056

 dense_8 (Dense)             (None, 1)                 33

=================================================================
Total params: 1377 (5.38 KB)
Trainable params: 1377 (5.38 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In this case, we will use `mean_squared_error` loss function. During training, the loss function will calculate the squared difference between the target value y and the predicted value $\hat{y}$, calculated as $\sum (y - \hat{y})^2$.

We can use the Adam optimizer for regression tasks as well. Note that we cannot measure accuracy as in classification tasks, therefore we don't need to specify a performance metric. Alternatively, we can use mean-squared error, mean-absolute error, or R2 score as metrics.

[64]:
```
# Compile the model
model_2.compile(loss='mean_squared_error', optimizer='adam')
```

Similarly to classification tasks, to fit the model we will provide the training data and the target values. Let's fit the model for 200 epochs.

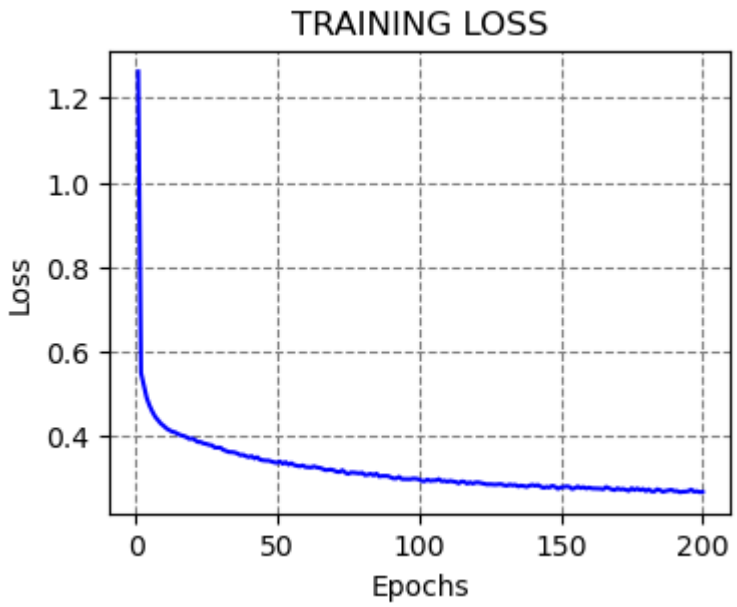
[65]:
```
history = model_2.fit(X_train_scaled, y_train, epochs=200, verbose=0)
```

[66]:
```
# plot the loss
train_loss = history.history['loss']

epochsn = np.arange(1, len(train_loss)+1,1)
plt.figure(figsize=(4, 3))

plt.plot(epochsn,train_loss, 'b')
plt.grid(color='gray', linestyle='--')
```

(continues on next page)

(continued from previous page)

```
plt.title('TRAINING LOSS')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```



We will again use `predict` to obtain the outputs of the model. Let's check the predicted values for the first 10 test samples.

[67]:
```
predictions = model_2.predict(X_test_scaled[:10])
predictions
```

```
1/1 [==============================] - 0s 52ms/step
```

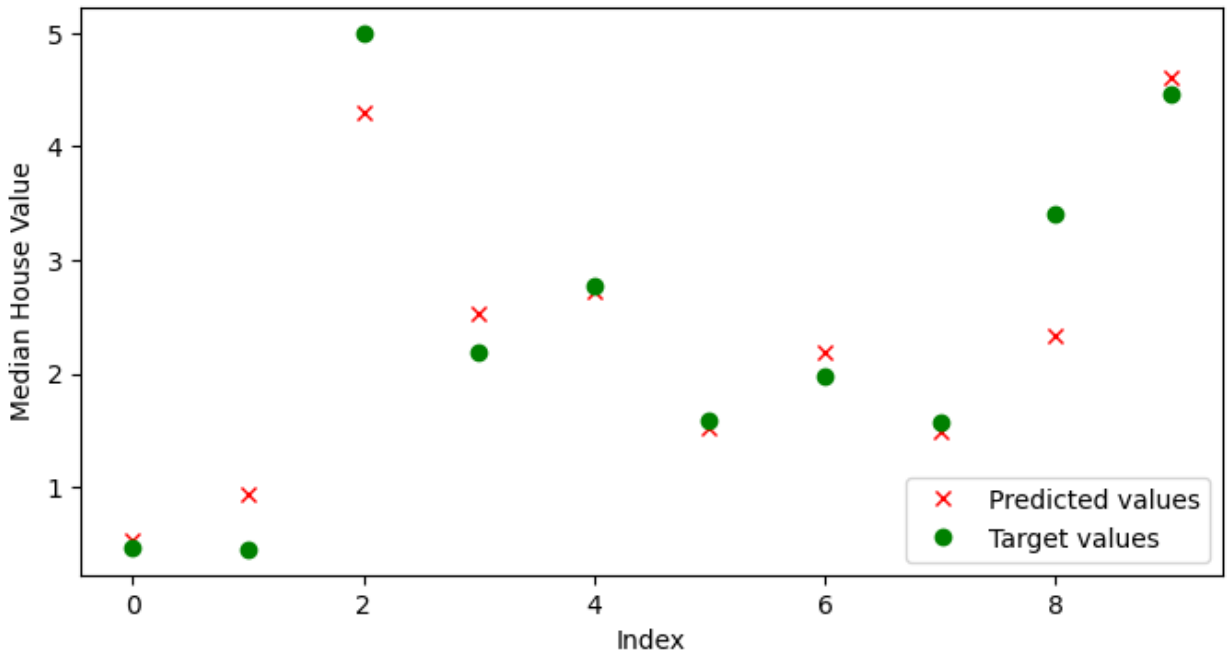
[67]:
```
array([[0.5398234],
       [0.9367044],
       [4.2926064],
       [2.5250819],
       [2.730356 ],
       [1.5289242],
       [2.1832035],
       [1.4888892],
       [2.338513 ],
       [4.607725 ]], dtype=float32)
```

And we can compare the predicted values by the model to the target ground-truth values. It would be easier to create a plot of the predicted and target values, as in the following cell. We can see that for most values, the model predictions are close to the target values.

[68]:
```
# Ground-truth values
np.array(y_test[:10])
```

[68]:
```
array([0.477  , 0.458  , 5.00001, 2.186  , 2.78   , 1.587  , 1.982  ,
       1.575  , 3.4    , 4.466  ])
```

[69]:
```
plt. figure(figsize=(8, 4))
plt.plot(predictions, 'rx', label='Predicted values')
plt.plot(np.array(y_test[:10]), 'go', label='Target values')
plt.legend()
plt.xlabel('Index')
plt.ylabel('Median House Value')
plt.show()
```



As we mentioned, metrics that we can use to evaluate the performance of regression models include mean-squared error (MSE), mean-absolute error (MAE), or R2 score. R2 score (or R^2 pronounced 'R squared') is also called coefficient of determination, and evaluates the goodness of fit of regression models, by measuring the proportion of the variation in the target variable that is contained in the predicted variable.

In the next cell we imported these metrics from scikit-learn and calculated their values. Note that for MSE and MAE lower values indicate better performance, and for R2 score greater values indicate better performance.

[70]:
```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

all_preds = model_2.predict(X_test_scaled)

print(f'MSE: {mean_squared_error(y_test, all_preds):7.4f}')
print(f'MAE: {mean_absolute_error(y_test, all_preds):7.4f}')
print(f'R2 score: {r2_score(y_test, all_preds):7.4f}')
```

```
129/129 [==============================] - 0s 1ms/step
MSE:  0.3119
MAE:  0.3678
R2 score:  0.7620
```

7.15.4 15.4 Saving and Loading Models in Keras

To save a model in Keras, we simply use the `save()` method, and provide the path to the directory.

The saved model contains the architecture with the layers, the weights for all layers in the model, and other information related to the used optimizer, loss, and metrics.

There are two possible formats to save the models into: HDF5 and Keras-TensorFlow SavedModel. If we provide the `.h5` extension, then the saved model format is HDF5. If we provide the `.keras` extension, or if we don't provide any extension, the model will be saved into Keras-TensorFlow SavedModel format.

Keras-TensorFlow SavedModel is the default, and it is recommended, since it is a newer format, and it saves additional information that is not included in the `h5` file, such as losses and metrics (e.g., they can be useful if we wish to resume training).

Let's use the Keras-TensorFlow SavedModel format to save the first classification model `model_1` from Section 15.2.1 in the current working directory under the name `classification_model`.

[71]:
```
# Saving a model
model_1.save('classification_model.keras')
```

To load a saved model we will use the `load_model` method. It returns the architecture and weights of the saved model.

Let's load the save model as `model_3`.

[72]:
```
# Loading a saved model
from keras.models import load_model

model_3 = load_model('classification_model.keras')
```

Afterward, we can use the loaded model as we would use any other model.

Let's evaluate the model on the test dataset, just to check whether the returned value would be as expected.

[73]:
```
# Evaluate on test dataset
evals_test = model_3.evaluate(test_data, test_labels)
print("Classification Accuracy: ", evals_test[1])
```

```
4/4 [==============================] - 0s 0s/step - loss: 0.0708 - accuracy: 0.9649
Classification Accuracy:  0.9649122953414917
```

The summary of the model is displayed below.

[74]:
```
model_3.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 30)]              0

 dense (Dense)               (None, 30)                930

 dense_1 (Dense)             (None, 15)                465

 dense_2 (Dense)             (None, 1)                 16

=================================================================
```

(continues on next page)

(continued from previous page)

```
Total params: 1411 (5.51 KB)
Trainable params: 1411 (5.51 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

7.15.5 Appendix

The material in the Appendix is not required for quizzes and assignments.

Keras-TensorFlow Model APIs

In Keras-TensorFlow, there are three ways to build ANNs:

- Functional API
- Sequential API
- Subclassing API

Functional API in Keras

In the above examples, we used Functional API to build our models. Repeated again in the next cell is the first model example for classification which we used in Section 5.2.1.

[75]:
```
# Define the layers in the network
inputs = Input(shape=(30,))
dense1 = Dense(units=30, activation='relu')(inputs)
dense2 = Dense(units=15, activation='relu')(dense1)
outputs = Dense(units=1, activation='sigmoid')(dense2)

# Define the model by providing the inputs and outputs
model_1 = Model(inputs, outputs)
```

Functional API is the recommended API, because it is a more recent and advanced API in comparison to Sequential API, but it is simpler than the Subclassing API.

Functional API allows to build models with multiple inputs and outputs, or with custom connections between the hidden layers (examples are residual or skip connections, which we will study in the following lectures). Because of these characteristics, Functional API is well suited for creating networks for advanced tasks, such as object detection, segmentation, and others.

Sequential API in Keras

Sequential API is the initial option in Keras for model building, as well as it is the simplest.

The corresponding sequential model to the above `model_1` is shown in the next cell. To create the model, we just add the layers in a sequence, one after another.

[76]:
```
from keras.models import Sequential

# Define the model and the layers in the network
```

(continues on next page)

(continued from previous page)

```
model_1_seq = Sequential([
        Dense(30, activation='relu'),
        Dense(15, activation='relu'),
        Dense(1, activation='sigmoid')])
```

After the model is created, the steps of compiling, training, and evaluation are the same, as shown in the next cell. As we explained above, the reason that the accuracy of `model_1_seq` is different than the accuracy of `model_1` is not because of the API used, but because of the stochastic nature of ANNs.

[77]:
```
# Compile the model
model_1_seq.compile(optimizer='adam',
                    loss='binary_crossentropy',
                    metrics=['accuracy'])

# Train the model
model_1_seq.fit(train_data, train_labels, epochs=100, batch_size=64, verbose=0)

# Evaluate on test dataset
evals_test_seq = model_1_seq.evaluate(test_data, test_labels)
print("Classification Accuracy: ", evals_test_seq[1])
```

```
4/4 [==============================] - 0s 0s/step - loss: 0.0672 - accuracy: 0.9825
Classification Accuracy:  0.9824561476707458
```

Sequential API is suitable for simple classification or regression tasks. However, this API is not suited for tasks with multiple inputs or outputs, or for networks with advanced architectures.

Subclassing API in Keras

Subclassing API is designed for building custom models and having full control of every step in the model building and training. It is the most suitable for building models with complex or custom architectures.

[78]:
```
# # Define the layers in the network
class Model1Subclass(Model):

    def __init__(self, **kwargs):
        super().__init__()
        self.dense_1 = Dense(30, activation='relu')
        self.dense_2 = Dense(15, activation='relu')
        self.dense_3 = Dense(1, activation='sigmoid')

    def call(self, inputs):
        x = self.dense_1(inputs)
        x = self.dense_2(x)
        x = self.dense_3(x)

    return x

# Instantiate the model
model_1_subclass = Model1Subclass()
```

[79]:
```
# Compile the model
model_1_subclass.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

# Train the model
model_1_subclass.fit(train_data, train_labels, epochs=100, batch_size=64, verbose=0)

# Evaluate on test dataset
evals_test_subclass = model_1_subclass.evaluate(test_data, test_labels)
print("Classification Accuracy: ", evals_test_subclass[1])
```

```
4/4 [==============================] - 0s 2ms/step - loss: 0.0725 - accuracy: 0.9825
Classification Accuracy:  0.9824561476707458
```

The Subclassing API in Keras is also similar to the model building in PyTorch. We will learn about PyTorch in an upcoming lecture.

7.15.6 References

1. Complete Machine Learning Package, Jean de Dieu Nyandwi, available at: https://github.com/Nyandwi/machine_learning_complete.
2. Implementing Linear Regression on California Housing Dataset, Debarshi Raj Basumatary, available at: https://medium.com/mlearning-ai/implementing-linear-regression-on-california-housing-dataset-378e14e421b7.

BACK TO TOP

7.16 Lecture 16 - Convolutional Neural Networks

7.16.1 16.1 Introduction to Convolutional Neural Networks

Convolutional Neural Networks (CNNs), also known as **ConvNets** have accelerated various computer vision tasks, such as image recognition and classification, image segmentation, and object detection. CNNs have been used in related applications, including autonomous vehicles, medical image diagnosis, intelligent robots, and others.

A typical architecture of CNN is shown below and consists of 3 main types of layers: convolutional layers, pooling layers, and fully-connected (dense) layers. CNNs typically have multiple blocks of convolutional and pooling layers, followed by fully-connected layers. The optimal number of layers is task-dependent, and it is a hyperparameter that needs to be tuned by the user during the training and model selection step.

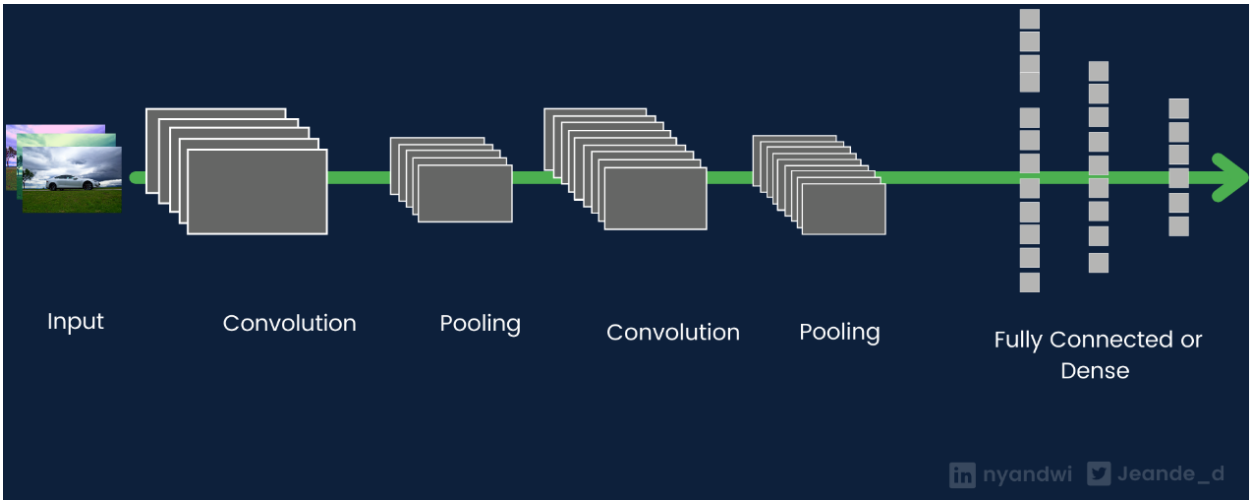


Figure: Architecture of a CNN.

Each layer in CNNs transforms the input images into a higher-level representation. The initial layers in CNNs learn low-level features (such as edges and lines), the middle layers learn mid-level features (e.g., textures and parts), and the last layers learn high-level features (such as objects). The fully-connected layers use the high-level features to classify the input image.

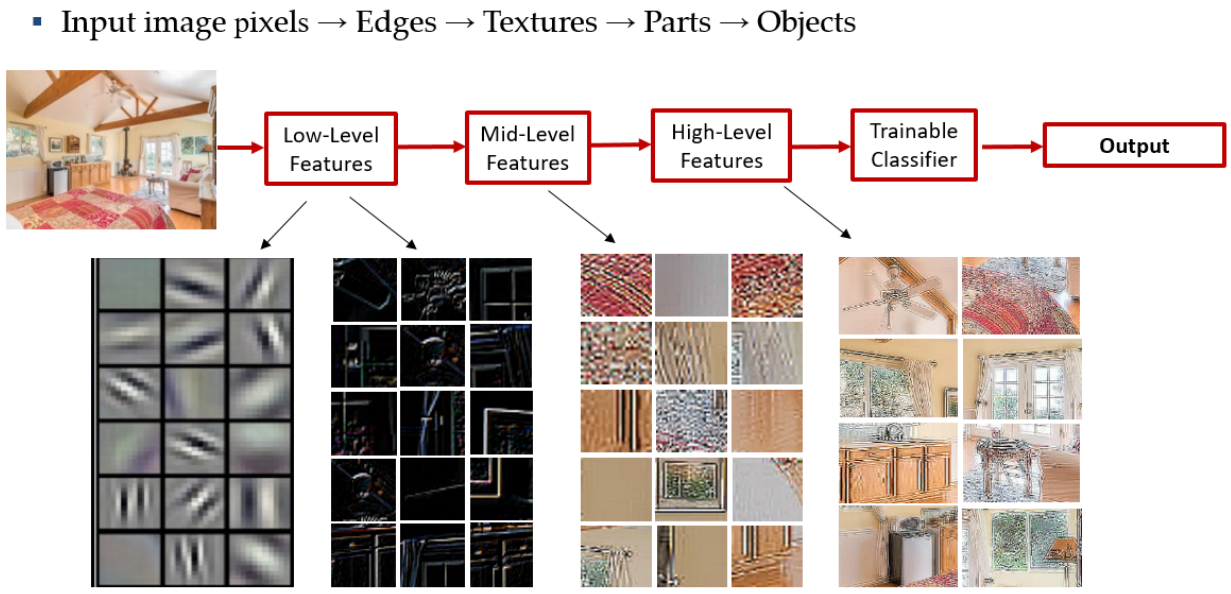


Figure: Extracted features in a CNN.

Convolutional Layers

The process of convolution is to pass a convolutional filter to each pixel in an image, multiply the corresponding pixels, and calculate the sum. This process is repeated until the filter is slid over all image pixels.

Figure: Convolution process.

Modern deep learning libraries, such as TensorFlow and PyTorch, allow to create convolutional layers in one line of code, as follows.

tf.keras.layers.Conv2D(…..)

Beside 2D convolutional filters that are used for processing images, 3D convolutional layers are used for analyzing videos, and 1D convolutional layers are used for analyzing time-series data.

The output of a convolutional layer is high-dimensional feature maps, and its dimension depends on the number of convolutional filters used in the layer. For example, if the layer has 32 filters, then we will have 32 feature maps at the output.

Pooling layers

Pooling layers are used to reduce the size of the feature maps output by convolutional layers, which reduces the number of network parameters and the computational cost for training the model. Pooling layers are also helpful for selecting more imporant information from the feature maps in the previous layer, they reduce the sensitivity of the model to the exact location of objects in inputs images, and they reduce the impact of noise in input data.

There are different pooling options, although the most common is the Maxpooling layer, which reduces the image sizes by keeping the pixels with the maximum intensity.

Implementing a pooling layer in most frameworks is very simple. The following figure depicts Maxpooling with a pool_size=2, which means that the output is the maximum value for each square of 2x2 pixels.

tf.keras.layers.MaxPooling2D(…)

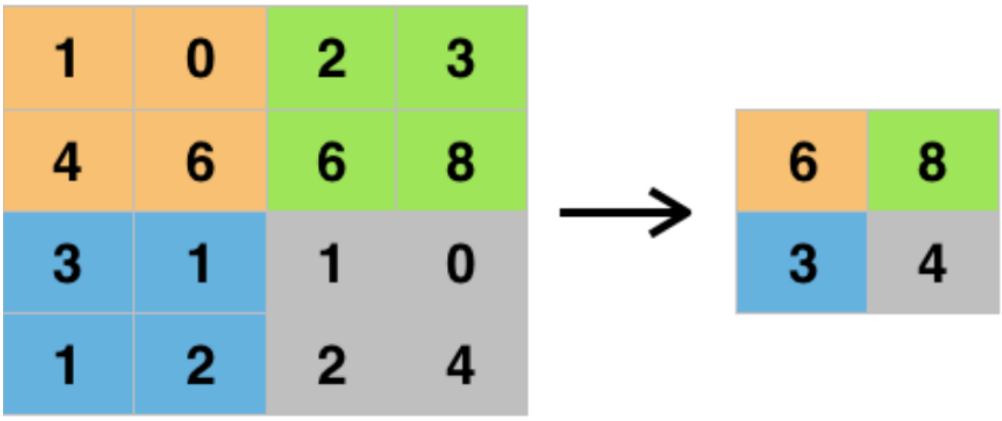


Figure: Pooling layer with a pool_size=2.

The portion of the network that consists of convolutional and pooling layers is often called an **encoder**, since it is used for encoding the information in input data into a representation that is more useful for the task at hand. Similarly, the portion of the network that consists of fully-connected layers is referred to as *classifier* (*head*, *top*) of the network.

Fully-connected Layers (Densely-connected Layers)

The last layers in ConvNets are fully-connected layers, that match the produced feature maps from the previous layers to the labels of the original image.

```
tf.keras.layers.Dense(....)
```

7.16.2 16.2 Loading the Dataset

In this notebook, we are going to use one of the most well-known datasets for image classification called `CIFAR-10`.

CIFAR-10 consists of 60,000 color images in 10 categories. The 10 classes are: `airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck`.

Each class contains 6,000 images. In the dataset, 50,000 images are allocated for training, and 10,000 for testing. The top accuracy on the test dataset is about 90%.

You can learn more about the dataset here.

Note that there is a larger version with 100 classes called CIFAR-100.

CIFAR-10 is available in Keras built-in datasets, so we can simply load the data by using the helper function `keras.datasets.cifar10.load_data()`.

[ ]:
```
# import required libraries
import tensorflow as tf
from tensorflow import keras

import numpy as np
import matplotlib.pyplot as plt
```

[ ]:
```
# Print the version of tf
print("TensorFlow version:{}".format(tf.__version__))
```

```
TensorFlow version:2.13.0
```

[ ]:
```
# Load the train and test images and labels
(train_data, train_label), (test_data, test_label) = keras.datasets.cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 3s 0us/step
```

As usual, we can inspect the shapes of the training and testing datasets.

Note that each image is a 32x32x3 tensor, having a size of 32x32 pixels and 3 channels corresponding to the RGB (red-green-blue) channels.

[ ]:
```
print('Training images', train_data.shape)
print('Training labels', train_label.shape)
print('Testing images', test_data.shape)
print('Testing labels', test_label.shape)
```

```
Training images (50000, 32, 32, 3)
Training labels (50000, 1)
Testing images (10000, 32, 32, 3)
Testing labels (10000, 1)
```

Data Preprocessing

The values for the pixel intensities in the images are in the range between 0 and 255. The type is `uint8` which stands for unsigned integer with 8 bits, that is, the possible values are between $2^0 = 1$ and $2^8 = 256$. This means that each pixel has an intensity value in that range, where 0 is a black pixel, 255 is a white pixel, and all other colors are in between.

[ ]:
```
# Display the range of images
print('Max pixel value', np.max(train_data))
print('Min pixel value', np.min(train_data))
print('Average pixel value', np.mean(train_data))
print('Data type', train_data[0].dtype)
```

```
Max pixel value 255
Min pixel value 0
Average pixel value 120.70756512369792
Data type uint8
```

When processing image data with neural networks, the pixel values are commonly normalized to the [0, 1] range. This allows the networks to train faster and it usually leads to better results. Let's normalize the images by dividing them with the maximum intensity of 255, and check again if the scaled data look correct. Note that the data type after the normalization is `float64`.

[ ]:
```
# Normalize the images
train_data = train_data / 255
test_data = test_data / 255
```

[ ]:
```
# Display the range of images (to make sure they are in the [0, 1] range)
print('Max pixel value', np.max(train_data))
print('Min pixel value', np.min(train_data))
print('Average pixel value', np.mean(train_data))
print('Data type', train_data[0].dtype)
```

```
Max pixel value 1.0
Min pixel value 0.0
Average pixel value 0.4733630004850874
Data type float64
```

The labels for the 10 classes in CIFAR-10 are shown below.

Label	Description
0	airplane
1	automobile
2	bird
3	cat
4	deer
5	dog
6	frog
7	horse
8	ship
9	truck

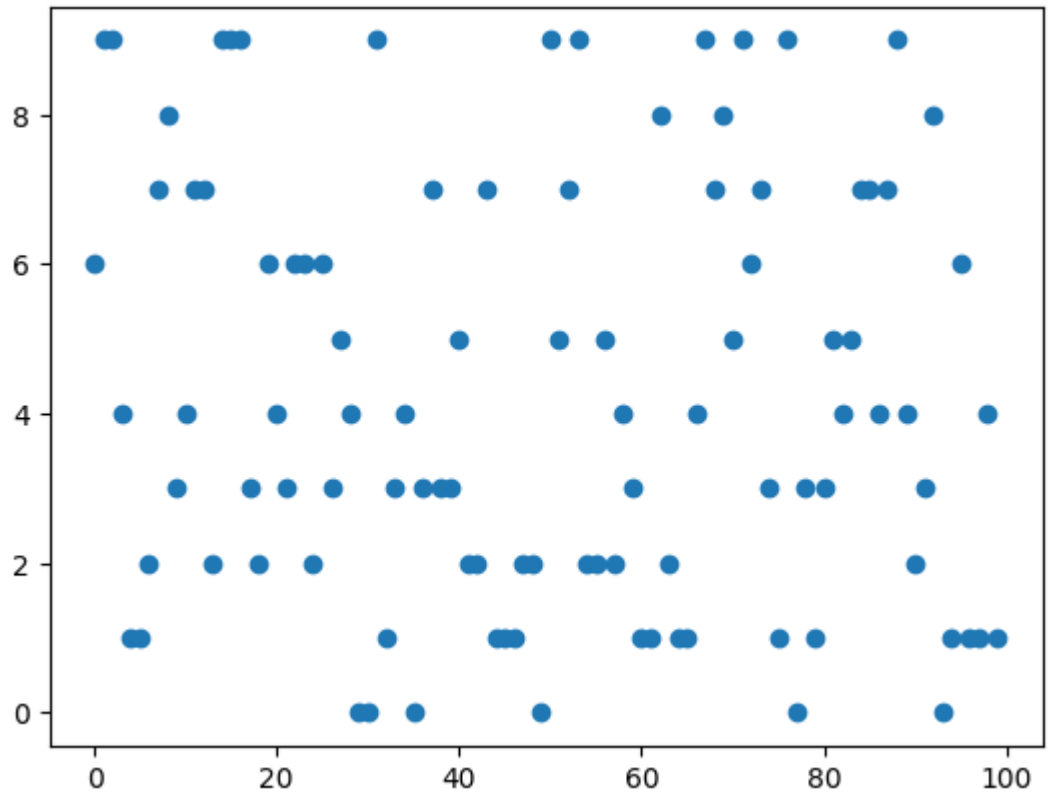
Let's inspect the first 10 values in the labels. We can see that each label corresponds to one of the 10 categories.

Also, in the next cell we plotted the first 100 labels, and we can notice that they have values between 0 and 9, so everything looks good.

[ ]:
```
print(train_label[:10])
```

```
[[6]
 [9]
 [9]
 [4]
 [1]
 [1]
 [2]
 [7]
 [8]
 [3]]
```

[ ]:
```
plt.plot(train_label[:100], 'o')
plt.show()
```



We can convert the labels to one-hot encoded values, by using the Keras function `to_categorical`. This transforms each label into a row of 10 values, since there are 10 classes. Note that for the first row now only the column with index 6 is 1, for the second row the last column is 1, etc., and all other values are 0.

[ ]:
```
# Convert the labels to one-hot encoding format
train_label_onehot = keras.utils.to_categorical(train_label, 10)
test_label_onehot = keras.utils.to_categorical(test_label, 10)

print('Labels train shape: {}'.format(train_label_onehot.shape))
```

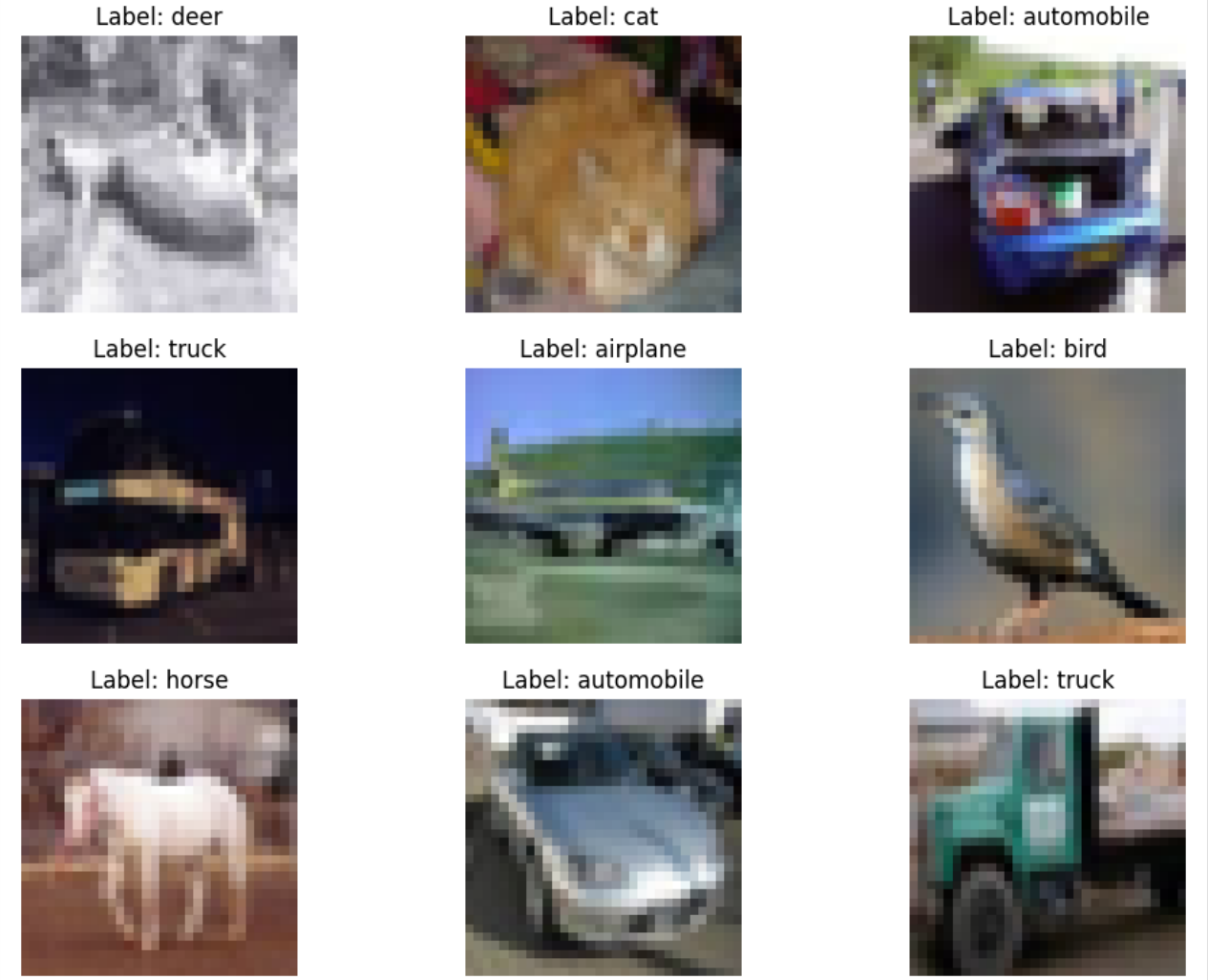
```
Labels train shape: (50000, 10)
```

[ ]:
```
# Check the labels
train_label_onehot[:5]
```

```
array([[0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]], dtype=float32)
```

Let's visualize some images to see what they look like. We can notice that the resolution is quite low, since they are 32x32 pixels.

[ ]:
```
label_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
↪'ship', 'truck']

# Plot a few images to check if the labels are correct
plt.figure(figsize=(12, 9))
for n in range(9):
    i = np.random.randint(0, len(train_data), 1)
    ax = plt.subplot(3, 3, n+1)
    plt.imshow(train_data[i[0]])
    plt.title('Label: ' + str(label_names[train_label[i[0]][0]]))
    plt.axis('off')
```



7.16.3 16.3 Creating, Training, and Evaluating a CNN Model

Create the CNN

Next, we are going to define the network architecture with the Keras library, which is a high-level API developed on top of TensorFlow. Recall from the previous lecture that the main data structures in Keras are **layers** and **models**.

In the first cell below we imported the layers, and afterward we defined the layers.

We need to first introduce an **Input** layer to provide the size of the data, and in this case, the shape of the input data is set to (32,32,3). This is because the images are arrays with shape 32x32x3.

Next, we will include a convolutional layer **Conv2D**. The arguments of Conv2D layer in Keras are:

- `filters`: define the number of filters in the layer.
- `kernel_size`: the height and width of each filter, defined as an integer such as `kernel_size=3`, or a tuple such as `kernel_size=(3,3)`.
- `padding`: it is not a very important argument, when it is equal to `'same'` it means that the output images are the same size as input images; otherwise, the output images can be slightly smaller.

Next, we will include a pooling layer **MaxPooling2D**. We can specify the pooling size with the `pool_size` argument, such as `pool_size=3`. Otherwise, the default pool size is 2.

ConvNets typically consist of several blocks of convolutional and pooling layers.

And **Fully Connected Layers** are used for matching the compressed feature maps to their labels.

And there is one more layer that is used called **Flatten**. This layer is needed because the outputs of the convolutional and maxpooling layers are 3-dimensional tensors, but the dense layers require one-dimensional data (vectors). The Flatten layer just concatenates (flattens) all dimensions of a tensor into an 1D array.

For example, the following layer that we defined below,

```
conv1b = Conv2D(filters=32, kernel_size=3, padding='same')(conv1a)
```

is a convolutional layer, which takes as input the previous layer named `conv1a`. The layer `conv1b` has 32 convolutional filters of size 3, and padding is applied to preserve the size of the images.

[ ]:
```
# import the layers and the model
from keras.models import Model
from keras.layers import Input
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
```

[ ]:
```
# Define the layers in the model
inputs = Input(shape=(32, 32, 3))
conv1a = Conv2D(filters=32, kernel_size=3, padding='same')(inputs)
conv1b = Conv2D(filters=32, kernel_size=3, padding='same')(conv1a)
pool1 = MaxPooling2D()(conv1b)
conv2a = Conv2D(filters=64, kernel_size=3, padding='same')(pool1)
conv2b = Conv2D(filters=64, kernel_size=3, padding='same')(conv2a)
pool2 = MaxPooling2D()(conv2b)
conv3a = Conv2D(filters=128, kernel_size=3, padding='same')(pool2)
conv3b = Conv2D(filters=128, kernel_size=3, padding='same')(conv3a)
pool3 = MaxPooling2D()(conv3b)
flat = Flatten()(pool3)
dense1 = Dense(128, activation='relu')(flat)
dense2 = Dense(64, activation='relu')(dense1)
outputs = Dense(10, activation='softmax')(dense2)

# Define the model with inputs and outputs
cifar_cnn = Model(inputs, outputs)
```

Also note that in multi-class classification problems, the last Dense layer has **softmax** activation function. Softmax activation outputs a multiclass probability distribution, that is, it computes the probability that an image belongs to one of the 10 classes.

After we define the layers, we create the model as an instance of the `Model` class, for which the arguments are the input and output layers.

We can inspect the architecture of the CNN with `cifar_cnn.summary()`.

[ ]:
```
# Model summary
cifar_cnn.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 conv2d (Conv2D)             (None, 32, 32, 32)        896

 conv2d_1 (Conv2D)           (None, 32, 32, 32)        9248

 max_pooling2d (MaxPooling2  (None, 16, 16, 32)        0
 D)

 conv2d_2 (Conv2D)           (None, 16, 16, 64)        18496

 conv2d_3 (Conv2D)           (None, 16, 16, 64)        36928

 max_pooling2d_1 (MaxPoolin  (None, 8, 8, 64)          0
 g2D)

 conv2d_4 (Conv2D)           (None, 8, 8, 128)         73856

 conv2d_5 (Conv2D)           (None, 8, 8, 128)         147584

 max_pooling2d_2 (MaxPoolin  (None, 4, 4, 128)         0
 g2D)

 flatten (Flatten)           (None, 2048)              0

 dense (Dense)               (None, 128)               262272

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 10)                650

=================================================================
Total params: 558186 (2.13 MB)
Trainable params: 558186 (2.13 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Compile the CNN

Next, we need to compile the model, and provide the loss function, optimizer, and metric.

From the previous lecture, we know that the following three crossentropy losses are commonly used in Keras:

- `binary_crosssentropy` used for binary classification (i.e., there are only 2 classes).
- `categorical_crossentropy` used for multiclass classification (3 and more classes) and the target labels are encoded in one-hot matrix format.
- `sparse_categorical_crossentropy` used for multiclass classification (3 and more classes) and the target labels are encoded in ordinal format.

Since we converted the target labels into one-hot encoding format, we will use categorical crossentropy.

Let's use the Adam optimizer, and accuracy metric.

[ ]:
```
cifar_cnn.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
```

Train the CNN

Training CNNs is similar to training Fully-connected NNs. We use the `fit` function and list the train data and labels, number of epochs, and batch size. Here we used a batch size of 128 images, hence, since the training dataset has 50,000 images, the updates of the model parameters will be repeated 50,000/128 = 391 times (notice this number under the Epoch in the cell output below). Since we selected 10 epochs, this means there will be 3,910 training steps in total.

[ ]:
```
cifar_cnn.fit(train_data, train_label_onehot, epochs=10, batch_size=128)
```

```
Epoch 1/10
391/391 [==============================] - 17s 13ms/step - loss: 1.4524 - accuracy: 0.
↪4773
Epoch 2/10
391/391 [==============================] - 5s 12ms/step - loss: 0.9526 - accuracy: 0.6673
Epoch 3/10
391/391 [==============================] - 5s 12ms/step - loss: 0.7646 - accuracy: 0.7346
Epoch 4/10
391/391 [==============================] - 6s 14ms/step - loss: 0.6408 - accuracy: 0.7783
Epoch 5/10
391/391 [==============================] - 6s 14ms/step - loss: 0.5399 - accuracy: 0.8123
Epoch 6/10
391/391 [==============================] - 5s 12ms/step - loss: 0.4562 - accuracy: 0.8411
Epoch 7/10
391/391 [==============================] - 7s 17ms/step - loss: 0.3865 - accuracy: 0.8641
Epoch 8/10
391/391 [==============================] - 5s 13ms/step - loss: 0.3291 - accuracy: 0.8851
Epoch 9/10
391/391 [==============================] - 5s 12ms/step - loss: 0.2797 - accuracy: 0.9020
Epoch 10/10
391/391 [==============================] - 5s 12ms/step - loss: 0.2475 - accuracy: 0.9130
```

```
<keras.src.callbacks.History at 0x78fa92328b20>
```

Evaluate on Test Data

We can first use the `predict` function to output the class for each image in the test dataset, and afterward use `accuracy_score` to calculate the accuracy.

[ ]:
```
from sklearn.metrics import accuracy_score

preds = cifar_cnn.predict(test_data)

accuracy = accuracy_score(test_label, np.argmax(preds, axis=1))
print('The test accuracy is {0:5.2f} %'.format(accuracy*100))
```

```
313/313 [==============================] - 1s 3ms/step
The test accuracy is 73.00 %
```

We can check that the shape of the predicted outputs is `(10000,10)`, since there are 10,000 test images and 10 classes,

Let's also display the predictions for the first 5 test images in the following cell. The model outputs probabilities for each of the 10 classes. The probabilities sum to 1. For instance, for the first test image, the model assigned the highest probability of 0.99 to the class with index 3.

[ ]:
```
# check the shape of the predictions
preds.shape
```

```
(10000, 10)
```

[ ]:
```
# display the predictions for the first 5 test images
print('Predictions for first 5 test images:\n', np.around(preds[:5],3))
```

```
Predictions for first 5 test images:
 [[0.    0.    0.    0.99  0.    0.01  0.    0.    0.    0.   ]
 [0.004 0.754 0.    0.    0.    0.    0.    0.    0.241 0.002]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.993 0.006]
 [0.825 0.    0.    0.    0.    0.    0.    0.    0.174 0.   ]
 [0.    0.    0.    0.    0.    0.    1.    0.    0.    0.   ]]
```

Next, let's output the indices with the highest probability for each image with `np.argmax`, and compare them to the ground-truth labels.

[ ]:
```
# print the index with highest probability for the first 5 test images
np.argmax(preds[:5], axis=1)
```

```
array([3, 1, 8, 0, 6])
```

[ ]:
```
# print the ground-truth label for the first 5 test images
test_label[:5]
```

```
array([[3],
       [8],
       [8],
       [0],
       [6]], dtype=uint8)
```

And, a simpler way to calculate only the accuracy is with `evaluate`.

[ ]:
```
# Evaluate on test dataset
evals_test = cifar_cnn.evaluate(test_data, test_label_onehot)
print("Classification Accuracy: ", evals_test[1])
```

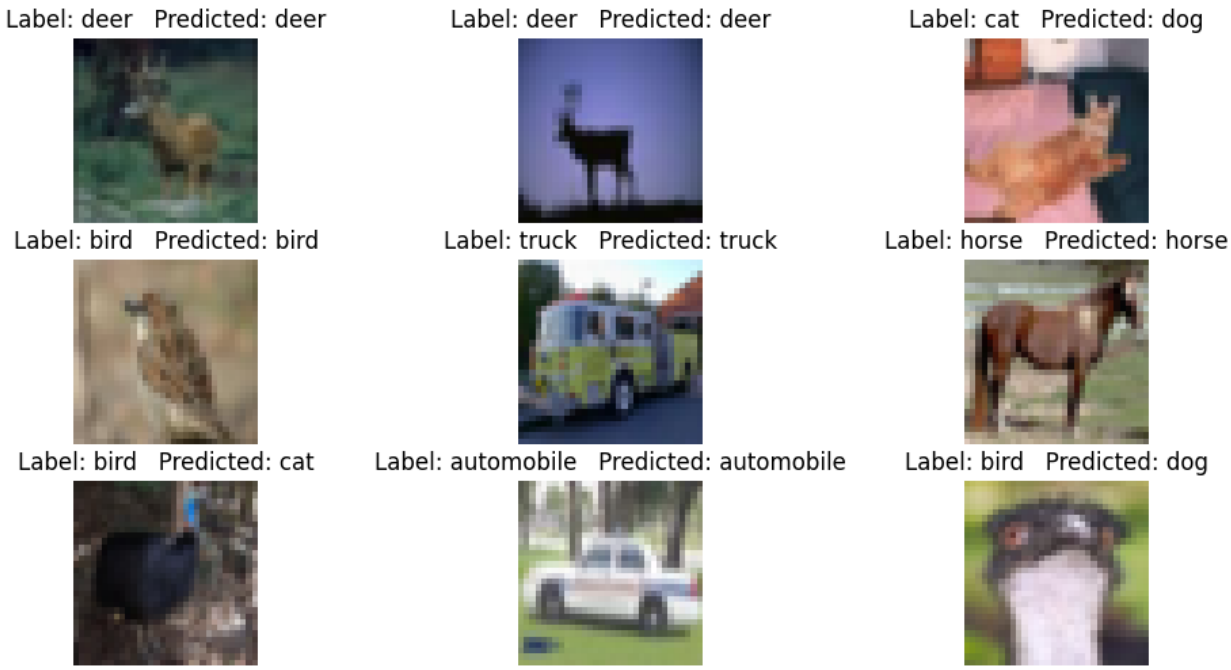
```
313/313 [==============================] - 1s 3ms/step - loss: 1.1302 - accuracy: 0.7300
Classification Accuracy:  0.7300000190734863
```

The important thing to notice is that the accuracy on the training dataset reached about 90%, while the accuracy on the test dataset is 73%. This means that the model overfits the training data. That is, the model begins to memorize the training data, and it learns to predict on the training dataset very well, but the generalization ability on unseen images is quite low. In the next sections we will learn how to deal with that.

Plot the Predictions

But first, let's plot a few images and the predicted class labels by the model.

```
[ ]: # Plot a few images to check them
plt.figure(figsize=(12, 6))
for n in range(9):
    i = np.random.randint(0, len(test_data), 1)
    ax = plt.subplot(3, 3, n+1)
    plt.imshow(test_data[i[0]])
    plt.title('Label: ' + str(label_names[test_label[i[0]][0]]) + '   Predicted: ' +␣
↪str(label_names[np.argmax(preds[i[0]])]))
    plt.axis('off')
```



7.16.4 16.4 Introduce a Validation Dataset

To be able to observe if the model overfits, one more dataset is introduced, referred to as a **validation dataset**. The original training dataset is typically randomly split, and approximately 70-80% is used as a training dataset, and 20-30% is used as a validation dataset.

This way, at the end of every epoch, we will calculate the accuracy of the model on the validation dataset. As the model is trained, if the training accuracy increases, but the validation accuracy decreases, it means that the model begins to overfit.

In Keras, the `fit` function has a `validation_split` argument, which allows us to use a percent of the training data for validation. In the next cell, we will use 20%, meaning that out of the 50,000 training images, the model will use 40,000 for training, and 10,000 for validation. An alternative way to introduce validation dataset is to first manually split the training dataset into two subsets of data, and in the `fit` function to pass the validation features and labels as a tuple named `validation_data`. However, the first option with using a `validation_split` argument is much simpler, and therefore preferred.

Note also that in the cell below we will just continue training the same model. If we re-defined and re-compiled the

model, we would have begun the training from scratch.

Now we can notice the overfitting, because the training accuracy continues to increase, but the validation accuracy decreases.

[ ]:
```
cifar_cnn.fit(train_data, train_label_onehot,
              epochs=10, batch_size=128,
              validation_split=0.2)
```

```
Epoch 1/10
313/313 [==============================] - 5s 17ms/step - loss: 0.2223 - accuracy: 0.
↪9208 - val_loss: 0.2551 - val_accuracy: 0.9123
Epoch 2/10
313/313 [==============================] - 4s 14ms/step - loss: 0.1890 - accuracy: 0.
↪9344 - val_loss: 0.3026 - val_accuracy: 0.8928
Epoch 3/10
313/313 [==============================] - 4s 13ms/step - loss: 0.1653 - accuracy: 0.
↪9440 - val_loss: 0.3451 - val_accuracy: 0.8859
Epoch 4/10
313/313 [==============================] - 4s 14ms/step - loss: 0.1461 - accuracy: 0.
↪9494 - val_loss: 0.3874 - val_accuracy: 0.8771
Epoch 5/10
313/313 [==============================] - 4s 13ms/step - loss: 0.1552 - accuracy: 0.
↪9463 - val_loss: 0.4818 - val_accuracy: 0.8587
Epoch 6/10
313/313 [==============================] - 4s 14ms/step - loss: 0.1367 - accuracy: 0.
↪9533 - val_loss: 0.4675 - val_accuracy: 0.8638
Epoch 7/10
313/313 [==============================] - 4s 14ms/step - loss: 0.1320 - accuracy: 0.
↪9553 - val_loss: 0.5117 - val_accuracy: 0.8595
Epoch 8/10
313/313 [==============================] - 4s 14ms/step - loss: 0.1268 - accuracy: 0.
↪9568 - val_loss: 0.6008 - val_accuracy: 0.8423
Epoch 9/10
313/313 [==============================] - 5s 14ms/step - loss: 0.1170 - accuracy: 0.
↪9603 - val_loss: 0.6020 - val_accuracy: 0.8495
Epoch 10/10
313/313 [==============================] - 4s 14ms/step - loss: 0.1050 - accuracy: 0.
↪9664 - val_loss: 0.6615 - val_accuracy: 0.8443
```

```
<keras.src.callbacks.History at 0x78f9ac92dfc0>
```

[ ]:
```
# Evaluate on test dataset
evals_test = cifar_cnn.evaluate(test_data, test_label_onehot)
print("Classification Accuracy: ", evals_test[1])
```

```
313/313 [==============================] - 1s 3ms/step - loss: 1.7219 - accuracy: 0.7239
Classification Accuracy:  0.7239000201225281
```

7.16.5 16.5 Dropout Layers

One way to deal with overfitting in neural networks is to introduce **Dropout** layers. The idea of dropout is very simple: during the training, at each step of processing a batch of images, a portion of the neurons in a layer are randomly disabled. This introduces randomness in the network, and it helps to improve model performance and reduce overfitting.

Dropout can be considered to be similar to the ensemble methods, where during each iteration, a slightly different model is trained, which uses only a portion of all neurons.

In the next cell, we add several dropout layers, where for instance, dropout of 0.2 means that 20% of the neurons in that layer are randomly dropped.

[ ]:
```
from keras.layers import Dropout

# Define the layers
inputs = Input(shape=(32, 32, 3))
conv1a = Conv2D(filters=32, kernel_size=3, padding='same')(inputs)
conv1b = Conv2D(filters=32, kernel_size=3, padding='same')(conv1a)
pool1 = MaxPooling2D()(conv1b)
dropout1 = Dropout(0.2)(pool1)
conv2a = Conv2D(filters=64, kernel_size=3, padding='same')(dropout1)
conv2b = Conv2D(filters=64, kernel_size=3, padding='same')(conv2a)
pool2 = MaxPooling2D()(conv2b)
dropout2 = Dropout(0.2)(pool2)
conv3a = Conv2D(filters=256, kernel_size=3, padding='same')(dropout2)
conv3b = Conv2D(filters=256, kernel_size=3, padding='same')(conv3a)
pool3 = MaxPooling2D()(conv3b)
flat = Flatten()(pool3)
dense1 = Dense(128, activation='relu')(flat)
dropout3 = Dropout(0.2)(dense1)
dense2 = Dense(64, activation='relu')(dropout3)
dropout4 = Dropout(0.2)(dense2)
outputs = Dense(10, activation='softmax')(dropout4)

# Define the model with inputs and outputs
cifar_cnn_2 = Model(inputs, outputs)
```

[ ]:
```
# compile model
cifar_cnn_2.compile(optimizer='adam',
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])

# fit model
history = cifar_cnn_2.fit(train_data, train_label_onehot,
                  epochs=40, batch_size=128,
                  validation_split=0.2, verbose=0)
```

[ ]:
```
# Evaluate on test dataset
evals_test = cifar_cnn_2.evaluate(test_data, test_label_onehot)
print("Classification Accuracy: ", evals_test[1])
```

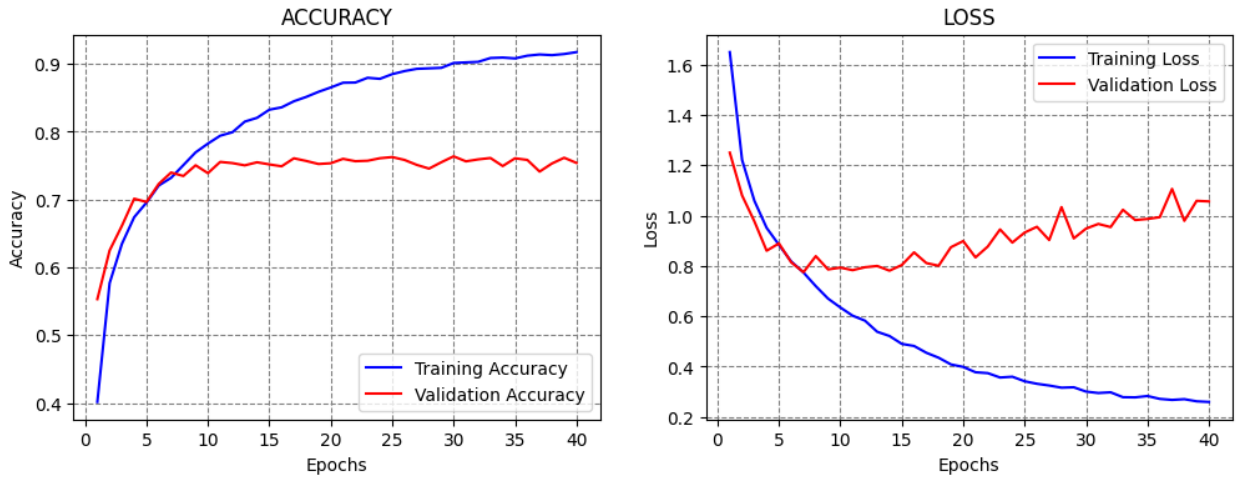
```
313/313 [==============================] - 1s 4ms/step - loss: 1.0808 - accuracy: 0.7516
Classification Accuracy:  0.7516000270843506
```

The performance was increased to about 75.1% from the initial 73% for the model without dropout layers.

Let's create a figure with the **learning curves**, to show the accuracy and loss of the model. The blue lines indicate the model performance on the training dataset and the red lines indicate the performance on the validation dataset.

We can notice that overfitting begins around epoch 15, when the training accuracy increases, but the validation accuracy stays about the same. Similarly, the training loss decreases, but the validation loss increases after epoch 15. The plots of the accuracy and loss are fairly correlated, however as we can see the accuracy can stay constant when the loss is changing.

[ ]:
```
# plot the accuracy and loss
train_loss = history.history['loss']
val_loss = history.history['val_loss']
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

epochsn = np.arange(1, len(train_loss)+1,1)
plt.figure(figsize=(12, 4))

plt.subplot(1,2,1)
plt.plot(epochsn, acc, 'b', label='Training Accuracy')
plt.plot(epochsn, val_acc, 'r', label='Validation Accuracy')
plt.grid(color='gray', linestyle='--')
plt.legend()
plt.title('ACCURACY')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.subplot(1,2,2)
plt.plot(epochsn,train_loss, 'b', label='Training Loss')
plt.plot(epochsn,val_loss, 'r', label='Validation Loss')
plt.grid(color='gray', linestyle='--')
plt.legend()
plt.title('LOSS')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```



7.16.6 16.6 Batch Normalization

Another type of layers that are used to reduce overfitting are **Batch Normalization** layers.

Similarly to scaling the input features to range [0,1] or to a Gaussian distribution with 0 mean and 1 standard deviation, Batch Normalization layers scale a batch of data to have 0 mean and 1 standard deviation. This reduces the instability in training the network due to differences among batches of data, and also reduces the overfitting.

[ ]:
```
from keras.layers import BatchNormalization

# Define the layers
inputs = Input(shape=(32, 32, 3))
conv1a = Conv2D(filters=32, kernel_size=3, padding='same')(inputs)
conv1b = Conv2D(filters=32, kernel_size=3, padding='same')(conv1a)
bn1 = BatchNormalization()(conv1b)
pool1 = MaxPooling2D()(bn1)
dropout1 = Dropout(0.2)(pool1)
conv2a = Conv2D(filters=64, kernel_size=3, padding='same')(dropout1)
conv2b = Conv2D(filters=64, kernel_size=3, padding='same')(conv2a)
bn2 = BatchNormalization()(conv2b)
pool2 = MaxPooling2D()(bn2)
dropout2 = Dropout(0.2)(pool2)
conv3a = Conv2D(filters=256, kernel_size=3, padding='same')(dropout2)
conv3b = Conv2D(filters=256, kernel_size=3, padding='same')(conv3a)
bn3 = BatchNormalization()(conv3b)
pool3 = MaxPooling2D()(bn3)
flat = Flatten()(pool3)
dense1 = Dense(128, activation='relu')(flat)
dropout3 = Dropout(0.2)(dense1)
dense2 = Dense(64, activation='relu')(dropout3)
dropout4 = Dropout(0.2)(dense2)
outputs = Dense(10, activation='softmax')(dropout4)

# Define the model with inputs and outputs
cifar_cnn_3 = Model(inputs, outputs)
```

[ ]:
```
# compile model
cifar_cnn_3.compile(optimizer='adam',
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])
```

Also, let's use the argument `verbose=0` to not display the loss and accuracy after every epoch. Instead, after the training is complete, we will display the learning curves to observe the performance of the model. We increased the number of epochs to 60 for this model.

And, we will use the `datatime` python library to measure and display the training time of the model.

[ ]:
```
import datetime
now = datetime.datetime.now

t = now()
# fit model
history = cifar_cnn_3.fit(train_data, train_label_onehot,
                  epochs=60, batch_size=128,
```

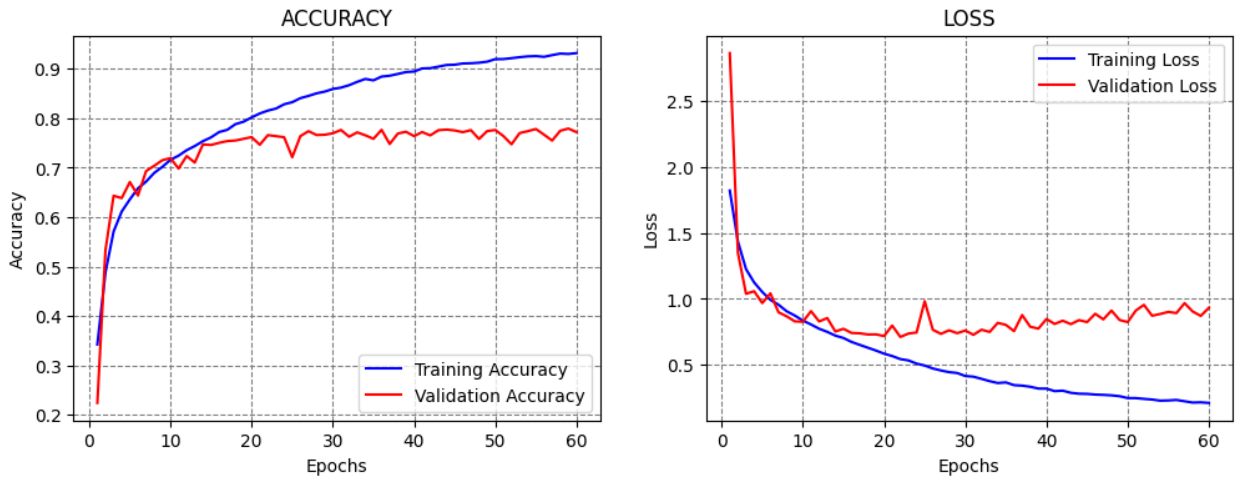
(continues on next page)

(continued from previous page)

```
                validation_split=0.2, verbose=0)

print('Training time: %s' % (now() - t))
```

```
Training time: 0:07:25.850362
```

[ ]:
```
# plot the accuracy and loss
train_loss = history.history['loss']
val_loss = history.history['val_loss']
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

epochsn = np.arange(1, len(train_loss)+1,1)
plt.figure(figsize=(12, 4))

plt.subplot(1,2,1)
plt.plot(epochsn, acc, 'b', label='Training Accuracy')
plt.plot(epochsn, val_acc, 'r', label='Validation Accuracy')
plt.grid(color='gray', linestyle='--')
plt.legend()
plt.title('ACCURACY')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.subplot(1,2,2)
plt.plot(epochsn,train_loss, 'b', label='Training Loss')
plt.plot(epochsn,val_loss, 'r', label='Validation Loss')
plt.grid(color='gray', linestyle='--')
plt.legend()
plt.title('LOSS')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```



[ ]:
```
# Evaluate on test dataset
evals_test = cifar_cnn_3.evaluate(test_data, test_label_onehot)
```

(continues on next page)

(continued from previous page)

```
print("Classification Accuracy: ", evals_test[1])
```

```
313/313 [==============================] - 1s 4ms/step - loss: 0.9479 - accuracy: 0.7658
Classification Accuracy:  0.7657999992370605
```

With the Batch Normalization layers in the model, the test accuracy increased to 76.6%.

7.16.7 16.7 Data Augmentation

One of the main reasons for overfitting is the lack of sufficient training samples, or lack of diversity in the training samples, for training a model.

To deal with overfitting we can use **data augmentation**, which refers to expanding the existing dataset by applying different image transformation operations. Examples are flipping the image vertically or horizontally, cropping the image, changing the contrast and color of the image, adding noise to the image, rotating the image at a given degree, and similar.

By doing data augmentation, we are synthesizing new images from existing data, as well as introducing some diversity in the images.

Keras provides the function `ImageDataGenerator` for data augmentation. Among the available operations for data augmentation are:

- **rotation_range** is a value in degrees (0–180) to randomly rotate images.

- **width_shift** and **height_shift** are ranges of fraction of total width or height within which to translate pictures, either vertically or horizontally.
- **shear_range** is for applying shearing randomly.
- **zoom_range** is for zooming in pictures randomly.
- **horizontal_flip** and **vertical_flip** are for flipping images.

In the example below we applied width shift, height shift, and horizontal flip.

[ ]:
```
from keras.preprocessing.image import ImageDataGenerator

# create data generator
datagen = ImageDataGenerator(width_shift_range=0.1, height_shift_range=0.1, horizontal_
↪flip=True)
```

ImageDataGenerator returns both the augmented images and their labels for each batch of images.

Let's plot a few images to see what they look like. They don't look much different, but that is because the applied shift and flip are not very large, and we didn't use other operations.

[ ]:
```
# Get images in batch of 20
aug_iterator = datagen.flow(train_data, train_label, batch_size=1)

plt.figure(figsize=(12,4))
for i in range(6):
        ax = plt.subplot(2, 3, i + 1)
        plt.imshow(aug_iterator[i][0][0])
        plt.title(str(label_names[aug_iterator[i][1][0][0]]))
        plt.axis("off")
```

When we use data augmentation, the arguments in the `fit` function are slightly different. First, we need to provide the name of our defined generator with `flow` to yield a batch of augmented images. In this case, we will need to use `datagen.flow`, since we assigned the image generator to the name `datagen`.

We also need to specify `steps_per_epoch`, which oftentimes is just the number of training images divided by the batch size.

Also, the generator in Keras does not accept a `validation_split` argument, and therefore we need to manually create a validation dataset, and then use it. This has been done in the next cells.

[ ]:
```
# Define the model with inputs and outputs
cifar_cnn_4 = Model(inputs, outputs)

# compile model
cifar_cnn_4.compile(optimizer='adam',
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])
```

[ ]:
```
# fit model
history = cifar_cnn_4.fit(datagen.flow(train_data, train_label, batch_size=128),
                steps_per_epoch=len(train_data)/128, epochs=120,
                validation_split=0.2, verbose=0)

# note that the generator does not work with validation split
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-40-843b623e8eaf> in <cell line: 2>()
      1 # fit model
----> 2 history = cifar_cnn_4.fit_generator(datagen.flow(train_data, train_label, batch_
↪size=128),
      3                 steps_per_epoch=len(train_data)/128, epochs=120,
      4                 validation_split=0.2, verbose=0)
      5

TypeError: Model.fit_generator() got an unexpected keyword argument 'validation_split'
```

[ ]:
```
from sklearn.model_selection import train_test_split

train_data_1, validation_data_1, train_label_1, validation_label_1 = train_test_
↪split(train_data, train_label_onehot, test_size=0.2, random_state=20, stratify=train_
↪label_onehot)
```

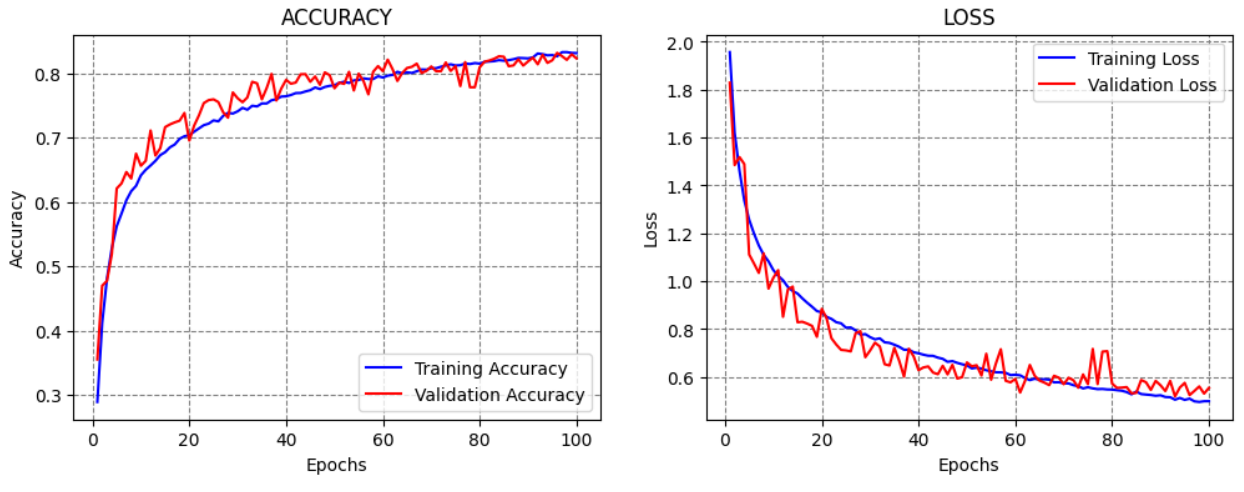
[ ]:
```
print('Training images', train_data_1.shape)
print('Training labels', train_label_1.shape)
print('Validation_images', validation_data_1.shape)
print('Validation labels', validation_label_1.shape)
```

```
Training images (40000, 32, 32, 3)
Training labels (40000, 10)
Validation_images (10000, 32, 32, 3)
Validation labels (10000, 10)
```

[ ]:
```
t = now()
# fit model
history= cifar_cnn_4.fit(datagen.flow(train_data_1, train_label_1, batch_size=128),
                steps_per_epoch=len(train_data_1)/128, epochs=100,
                validation_data=(validation_data_1, validation_label_1), verbose=0)

print('Training time: %s' % (now() - t))
```

```
Training time: 0:48:39.114947
```

[ ]:
```
# plot the accuracy and loss
train_loss = history.history['loss']
val_loss = history.history['val_loss']
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

epochsn = np.arange(1, len(train_loss)+1,1)
plt.figure(figsize=(12, 4))

plt.subplot(1,2,1)
plt.plot(epochsn, acc, 'b', label='Training Accuracy')
plt.plot(epochsn, val_acc, 'r', label='Validation Accuracy')
plt.grid(color='gray', linestyle='--')
plt.legend()
plt.title('ACCURACY')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.subplot(1,2,2)
plt.plot(epochsn,train_loss, 'b', label='Training Loss')
plt.plot(epochsn,val_loss, 'r', label='Validation Loss')
plt.grid(color='gray', linestyle='--')
plt.legend()
plt.title('LOSS')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```



[ ]:
```
# Evaluate on test dataset
evals_test = cifar_cnn_4.evaluate(test_data, test_label_onehot)
print("Classification Accuracy: ", evals_test[1])
```

```
313/313 [==============================] - 1s 4ms/step - loss: 0.5681 - accuracy: 0.8155
Classification Accuracy:  0.815500020980835
```

The data augmentation was helpful to further improve the performance of the model to about 81.5% accuracy. We trained the model for 100 epochs, which took about 48 minutes. From the learning curves we can conclude that the accuracy was still increasing at epoch 100, therefore we could continue training the model, or we could have selected

a higher number of epochs initially.

7.16.8 16.8 Transfer Learning

Transfer learning uses pretrained models that are trained on very large datasets to improve the performance on smaller datasets.

There are many pretrained models available in Keras Applications.

Typical steps in transfer learning include:

- Initialize the pretrained model (base model) and weights, and remove the top layers (fully-connected layers) of the pretrained model.
- Create a new model by adding new trainable fully-connected layers on top of the primary model.
- Train the new model, and evaluate the performance.

In this case we will use the VGG16 model, which is popular for image classification. The model is imported with the weights pretrained on ImageNet, which is a very large dataset with 1.2 million images in 1,000 classes. Since our task has only 10 classes, we will remove the dense layers which are very specific to ImageNet's 1,000 classes, and add our own custom layers with 10 classes.

Transfer learning is also often referred to as **model fine-tuning**, because we start with a pretrained model and we fine-tune its parameters to our custom dataset.

[ ]:
```
from tensorflow.keras.applications import vgg16
from keras.layers import GlobalAveragePooling2D

base_model = vgg16.VGG16(weights='imagenet', include_top=False, input_shape=(32,32,3))

# Add a global spatial average pooling layer
x = base_model.output
x = GlobalAveragePooling2D()(x)

x = Dense(128, activation='relu')(x)
x = Dropout(0.2)(x)
x = Dense(64, activation='relu')(x)
x = Dropout(0.2)(x)
predictions = Dense(10, activation='softmax')(x)

# The model we will train
cifar_cnn_5 = Model(inputs=base_model.input, outputs=predictions)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/
↪vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 [==============================] - 0s 0us/step
```

Early Stopping

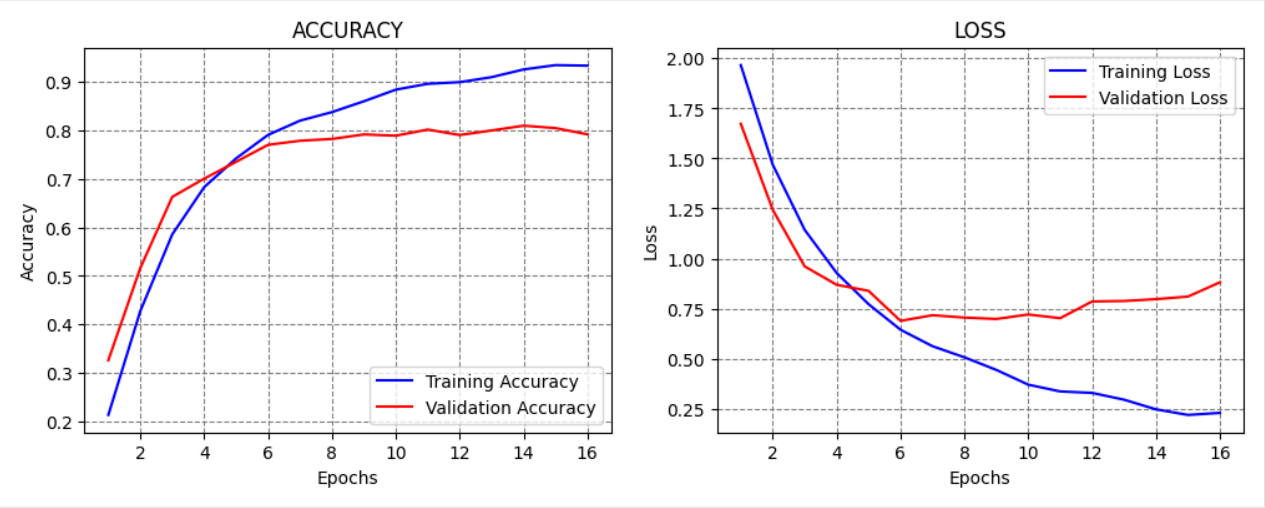
Keras has implemented several callbacks that allow to monitor the training process and have more control over it. We will study the various callbacks in the next lecture, and in this lecture we we explain how the callback Early Stopping works.

EarlyStopping monitors a metric (e.g., validation loss) and if the metric doesn't improve for a certain number of epochs (a.k.a. patience), terminates the training. By improve, we mean the decrease if the metric is loss, or increase if the metric is accuracy.

[ ]:
```
from keras.callbacks import EarlyStopping

cifar_cnn_5.compile(optimizer='adam', loss='categorical_crossentropy', metrics = [
↪'accuracy'])

# fit model
t = now()
history = cifar_cnn_5.fit(train_data, train_label_onehot, validation_split=0.2, batch_
↪size=128,
                    epochs=100, verbose=0, callbacks=[EarlyStopping(monitor='val_loss',␣
↪patience = 10)])

print('Training time: %s' % (now() - t))
```

```
Training time: 0:04:40.541954
```

[ ]:
```
# plot the accuracy and loss
train_loss = history.history['loss']
val_loss = history.history['val_loss']
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

epochsn = np.arange(1, len(train_loss)+1,1)
plt.figure(figsize=(12, 4))

plt.subplot(1,2,1)
plt.plot(epochsn, acc, 'b', label='Training Accuracy')
plt.plot(epochsn, val_acc, 'r', label='Validation Accuracy')
plt.grid(color='gray', linestyle='--')
plt.legend()
plt.title('ACCURACY')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.subplot(1,2,2)
plt.plot(epochsn,train_loss, 'b', label='Training Loss')
plt.plot(epochsn,val_loss, 'r', label='Validation Loss')
plt.grid(color='gray', linestyle='--')
plt.legend()
plt.title('LOSS')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```



[ ]:
```
# Evaluate on test dataset
evals_test = cifar_cnn_5.evaluate(test_data, test_label_onehot)
print("Classification Accuracy: ", evals_test[1])
```

```
313/313 [==============================] - 3s 9ms/step - loss: 0.9085 - accuracy: 0.7865
Classification Accuracy:  0.7864999771118164
```

Although the accuracy is reduced, note that the model was trained in about 15 epochs, in comparison to 100 epochs or more epochs when trained from scratch.

The following figure illustrates the early stopping callback. When the validation loss starts increasing, it terminates the training. In our case, the validation loss reached minimum at epoch 6, and since we specified a 10 epochs patience, the model stopped at epoch 16.

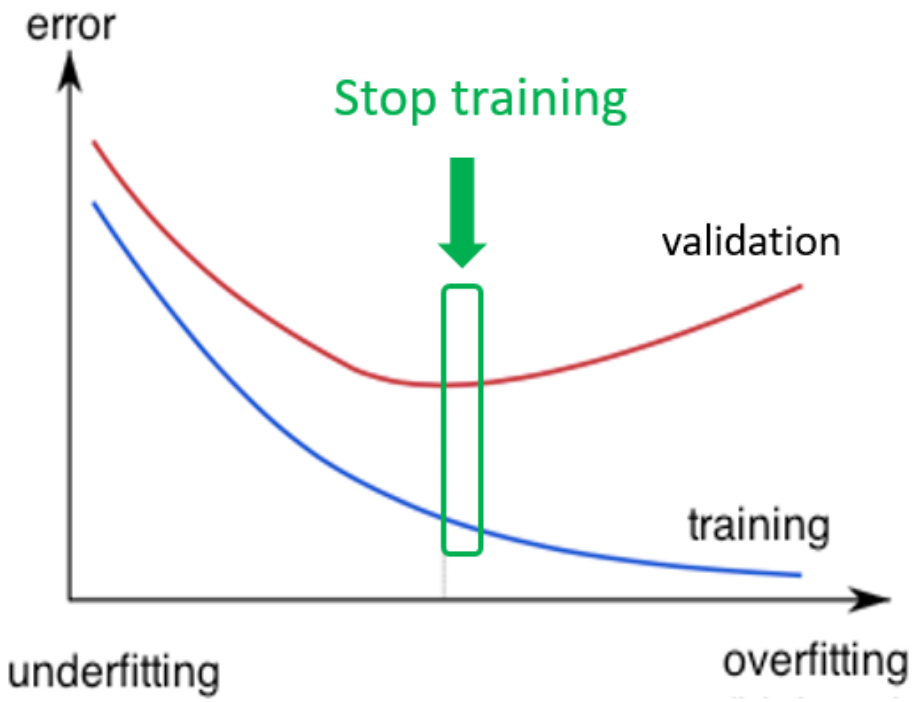


Figure: Early stopping.

Let's try to use a pretrained model and combine it with data augmentation and early stopping.

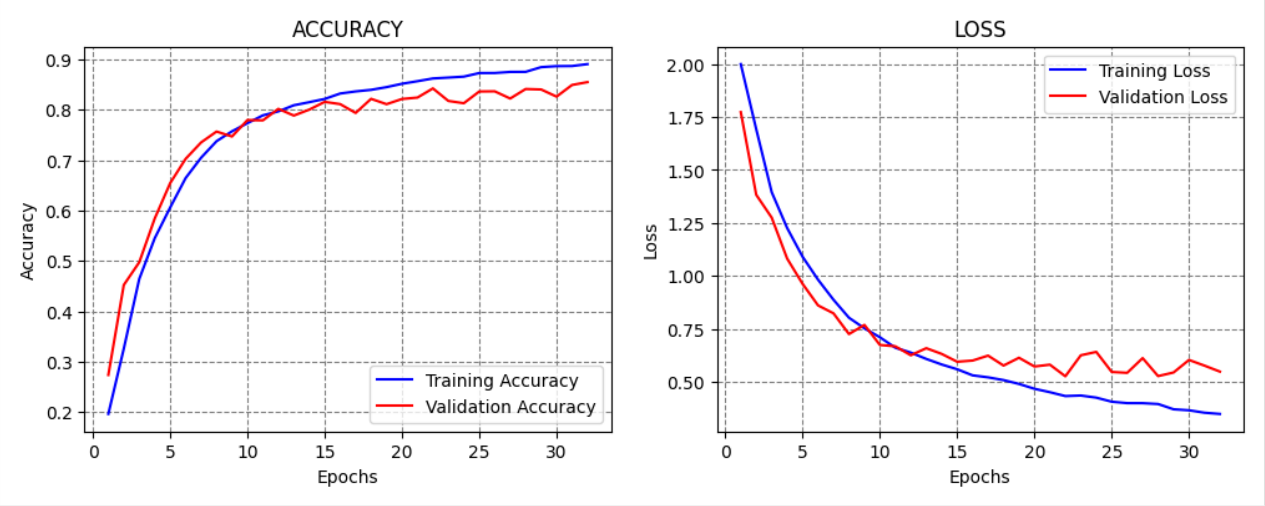
[ ]:
```
# The model we will train
cifar_cnn_6 = Model(inputs=base_model.input, outputs=predictions)
```

[ ]:
```
# compile model
cifar_cnn_6.compile(optimizer='adam',
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])
```

[ ]:
```
t = now()
# fit model
history= cifar_cnn_6.fit(datagen.flow(train_data_1, train_label_1, batch_size=128),
                 steps_per_epoch=len(train_data_1)/128, epochs=100,
                 validation_data=(validation_data_1, validation_label_1), verbose=0,
                 callbacks=[EarlyStopping(monitor='val_loss', patience = 10)])

print('Training time: %s' % (now() - t))
```

```
Training time: 0:16:44.402687
```

[ ]:
```
# plot the accuracy and loss
train_loss = history.history['loss']
val_loss = history.history['val_loss']
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

epochsn = np.arange(1, len(train_loss)+1,1)
plt.figure(figsize=(12, 4))

plt.subplot(1,2,1)
plt.plot(epochsn, acc, 'b', label='Training Accuracy')
plt.plot(epochsn, val_acc, 'r', label='Validation Accuracy')
plt.grid(color='gray', linestyle='--')
plt.legend()
plt.title('ACCURACY')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.subplot(1,2,2)
plt.plot(epochsn,train_loss, 'b', label='Training Loss')
plt.plot(epochsn,val_loss, 'r', label='Validation Loss')
plt.grid(color='gray', linestyle='--')
plt.legend()
plt.title('LOSS')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```



[ ]:
```
# Evaluate on test dataset
evals_test = cifar_cnn_6.evaluate(test_data, test_label_onehot)
print("Classification Accuracy: ", evals_test[1])
```

```
313/313 [==============================] - 3s 9ms/step - loss: 0.5882 - accuracy: 0.8575
Classification Accuracy:  0.8575000166893005
```

Combining transfer learning with data augmentation increased the accuracy to over 85%.

7.16.9 References

1. Complete Machine Learning Package, Jean de Dieu Nyandwi, available at: https://github.com/Nyandwi/machine_learning_complete.
2. How to Develop a CNN from Scratch for CIFAR-10 Photo Classification, Jason Brownlee, available at: https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-cifar-10-photo-classification/.
3. Python Machine Learning (2nd Ed.) Code Repository, Sebastian Raschka, available at: https://github.com/rasbt/python-machine-learning-book-2nd-edition.

BACK TO TOP

7.17 Lecture 17 - Model Selection, Hyperparameter Tuning

7.17.1 17.1 Model Selection

Model selection in Machine Learning is selecting one final model for a given task, e.g., that will be deployed in production. In general, selecting the "best" model should be based not only on the obtained values of relevant performance metrics (accuracy, specificity, sensitivity), but also based on other considerations, such as computational expense, available resources, model complexity, maintainability, and similar.

An important phase in selecting a candidate Machine Learning model is **hyperparameter tuning**. In the previous lecture on scikit-learn, we mentioned that **parameters (weights)** of the model have values that are updated in an iterative process during training, whereas **hyperparameters** (tuning parameters) are a set of parameters that control the complexity and performance of the model, and are selected (tuned) by the user. The hyperparameters are not updated during model training, and they stay constant. **Hyperparameter tuning** (also known as hypertuning) is the process of selecting hyperparameter values to find a model that generalizes well to unseen data. In the previous lectures, we explained that scikit-learn offers functions for Grid Search and Random Search, which search for solutions over different values of the hyperparameters and select an optimal set of hyperparameters for a given performance metric.

Another important aspect for model selection is the evaluation of model performance. It is particularly important to evaluate candidate models on unseen data during training, and this is typically achieved by splitting the available data into training and test datasets. We also learned that k-fold cross-validation can be used to draw folds from the available data, and evaluate the models on multiple different folds by resampling the available data. With k-fold cross-validation, each data point can appear only in one of the folds when evaluating the model performance.

Model selection in general can involve various data preprocessing techniques, evaluating different feature engineering strategies, applying Ensemble Methods to aggregate the predictions from several individual Machine Learning models, etc.

In this lecture, we will focus on hyperparameter tuning with neural networks. Namely, neural networks are more sensitive to hyperparameter tuning than conventional Machine Learning models (such as Liner Regression, k-Nearest Neighbors, etc.). We saw in the previous lectures that even if we use default values for the models in scikit-learn without any hyperparameter tuning, the models can still achieve solid performance. This is rarely the case with neural networks, as they usually require at least some hyperparameter tuning.

One note is to not confuse *hyperparameter tuning* with *model fine-tuning*, which refers to using a pretrained model on a large dataset and fine-tuning the model parameters on a smaller dataset.

Hyperparameters in Neural Networks

Let's examine again the ConvNet that we used in a previous lecture to classify images in the CIFAR-10 dataset, and let's try to identify the hyperparameters of the model.

```
# define the layers in the model
inputs = Input(shape=(32, 32, 3))
conv1a = Conv2D(filters=32, kernel_size=3, padding='same')(inputs)
conv1b = Conv2D(filters=32, kernel_size=3, padding='same')(conv1a)
pool1 = MaxPooling2D()(conv1b)
conv2a = Conv2D(filters=64, kernel_size=3, padding='same')(pool1)
```

(continues on next page)

(continued from previous page)

```
conv2b = Conv2D(filters=64, kernel_size=3, padding='same')(conv2a)
pool2 = MaxPooling2D()(conv2b)
conv3a = Conv2D(filters=128, kernel_size=3, padding='same')(pool2)
conv3b = Conv2D(filters=128, kernel_size=3, padding='same')(conv3a)
pool3 = MaxPooling2D()(conv3b)
flat = Flatten()(pool3)
dense1 = Dense(128, activation='relu')(flat)
dropout1 = Dropout(0.25)(dense1)
dense2 = Dense(64, activation='relu')(dropout1)
dropout2  = Dropout(0.25)(dense2)
outputs = Dense(10, activation='softmax')(dropout2)

# define the model with inputs and outputs
cifar_cnn = Model(inputs, outputs)

# compile the model
cifar_cnn.compile(optimizer=Adam(learning_rate=1e-3), loss='categorical_crossentropy', ␣
↪metrics=['accuracy'])

# train the model
cifar_cnn.fit(train_data, train_label_onehot, epochs=10, batch_size=128)
```

Hyperparameters in the above model include:

- Learning rate of the optimizer
- Batch size
- Number of training epochs
- Number of Convolutional layers
- Number of convolutional filters in the Convolutional layers
- Kernel size of the convolutional filters
- Type of padding in the Convolutional layers
- Number of Dense layers
- Number of neurons in each Dense layer
- Type of activation functions used in the layers
- Number of Dropout layers
- Dropout rate in the Dropout layers
- Type of optimizer (e.g., Adam, SGD, Nadam, RMSProp)
- Other parameters used in the optimizer (e.g., momentum)
- Type of initialization for the parameters in the model

There can be other hyperparameters depending on the network, however, one immediate observation is that neural networks have a large number of hyperparameters, and tuning all hyperparameters can be challenging as it may take significant time and resources.

On the other hand, not all of the hyperparameters have significant impact on the performance of the model. Out of all hyperparameters, probably the most important is the learning rate, and in most cases, some tuning of the learning rate

is required. In this lecture we will present techniques for hyperparameter tuning of a ConvNet model built with Keras and TensorFlow.

7.17.2 17.2 Evaluate the Impact of the Learning Rate

Loading a Custom Image Dataset in Keras

In previous lectures we worked with datasets that are built-in in Keras or scikit-learn and that can be directly loaded. Let's look at loading a custom dataset that is not part of the popular ML libraries. The dataset is saved in a folder on my Google Drive, therefore I need to first mount the Google Drive in order to access the folder with the images.

For this lecture, we will use the LFW dataset (Labeled Faces in the Wild), which consists of about 5,000 images of 62 celebrities. The next cells load the dataset and plot a few images to make sure that the labels are correct.

[ ]:
```
# mount the google drive
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.
↪mount("/content/drive", force_remount=True).
```

[ ]:
```
# import packages
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from keras.utils import load_img, img_to_array
import os
from os import listdir
import csv
import natsort

# Print the version of tf
print("TensorFlow version:{}".format(tf.__version__))
```

```
TensorFlow version:2.14.0
```

[ ]:
```
# Path to the directory containing the dataset
!unzip -uq 'drive/MyDrive/Data_Science_Course/Fall_2023/Lecture_17-Model_Selection,
↪Tuning/data/LFW-dataset.zip' -d 'sample_data/'
```

[ ]:
```
# Directories
train_dir = 'sample_data/LFW-dataset/Train/'
test_dir = 'sample_data/LFW-dataset/Test/'
val_dir = 'sample_data/LFW-dataset/Validation/'
labels_dir = 'sample_data/LFW-dataset/'

# Size of images (pixel width and height)
image_size = 100

# Function for loading the images
def load_imgs(path):
    # List of all images in the folder
```

(continues on next page)

(continued from previous page)

```
    imgList = listdir(path)
    # Make sure that the images are sorted in ascending order
    imgList=natsort.natsorted(imgList)
    # Number of images
    number_imgs = len(imgList)
    # Initialize numpy arrays for the images
    images = np.zeros((number_imgs, image_size, image_size, 3))
    # Read the images
    for i in range(number_imgs):
        tmp_img = load_img(path + imgList[i], target_size=(image_size, image_size, 3))
        img = img_to_array(tmp_img)
        images[i] = img/255.0
    return images

# Call the above function to load the images as numpy arrays
imgs_train = load_imgs(train_dir)
imgs_test = load_imgs(test_dir)
imgs_val = load_imgs(val_dir)
```

[ ]:
```
# Load the labels as numpy arrays
labels_train = np.genfromtxt(labels_dir + "train_labels.csv", delimiter=',', dtype=np.
↪int32)
labels_test = np.genfromtxt(labels_dir + "test_labels.csv", delimiter=',', dtype=np.
↪int32)
labels_val = np.genfromtxt(labels_dir + "val_labels.csv", delimiter=',', dtype=np.int32)
```

[ ]:
```
# Display the shapes of train, validation, and test datasets
print('Images train shape: {} - Labels train shape: {}'.format(imgs_train.shape, labels_
↪train.shape))
print('Images validation shape: {} - Labels validation shape: {}'.format(imgs_val.shape,␣
↪labels_val.shape))
print('Images test shape: {} - Labels test shape: {}'.format(imgs_test.shape, labels_
↪test.shape))

# Display the range of images (to make sure they are in the [0, 1] range)
print('\nMax pixel value', np.max(imgs_train))
print('Min pixel value', np.min(imgs_train))
print('Average pixel value', np.mean(imgs_train))
print('Data type', imgs_train[0].dtype)
```

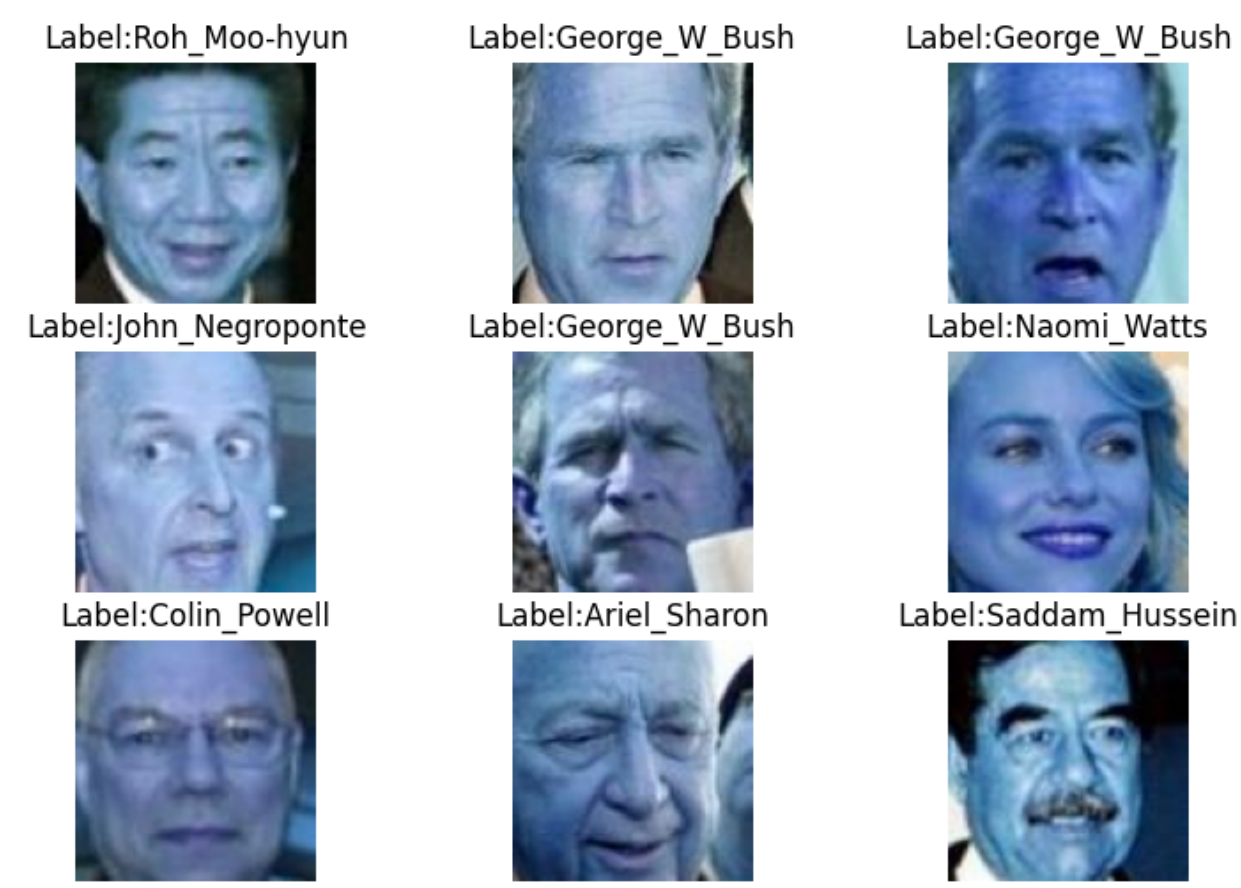
```
Images train shape: (3043, 100, 100, 3) - Labels train shape: (3043,)
Images validation shape: (1021, 100, 100, 3) - Labels validation shape: (1021,)
Images test shape: (1049, 100, 100, 3) - Labels test shape: (1049,)

Max pixel value 1.0
Min pixel value 0.0
Average pixel value 0.47390351913451484
Data type float64
```

[ ]:
```
# Read the names of the celebrities in the dataset (there are 62 celebrities)
name_list = []
with open(labels_dir+'name_list.csv', 'r') as f:
```

(continues on next page)

(continued from previous page)

```
    reader = csv.reader(f)
    for row in reader:
        name_list.append(row[1])

# Plot a few images to check if the the labels are correct
# There are a few bad images in the dataset, it needs to be cleaned
plt.figure(figsize=(9, 6))
for n in range(9):
    i = np.random.randint(0, len(imgs_train), 1)
    ax = plt.subplot(3, 3, n+1)
    plt.imshow(imgs_train[i[0]])
    plt.title('Label:' + str(name_list[labels_train[i[0]][]]))
    plt.axis('off')
```



Define the Model

We will use a pretrained VGG-16 model, and we will just add a classifier with 3 Dense layers on top of the model to fine-tune it to the LFW dataset.

[ ]:
```
from keras.models import Model
from keras.layers import Dense, GlobalAveragePooling2D, Dropout
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau,␣
↪LearningRateScheduler
from keras.applications import vgg16

import datetime
now = datetime.datetime.now
```

[ ]:
```
def Network():

    base_model = vgg16.VGG16(weights='imagenet', include_top=False, input_shape=(image_
↪size, image_size, 3))

    # Add a global spatial average pooling layer
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    # Add fully-connected layers
    x = Dense(1024, activation='relu')(x)
    x = Dropout(0.25)(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(0.25)(x)
    # Add a softmax layer
    predictions = Dense(62, activation='softmax')(x)

    # The model
    model = Model(inputs=base_model.input, outputs=predictions)

    return model
```

Let's define a function for plotting the accuracy and loss called `plot_accuracy_loss`, which we can call with different models to examine the learning curves.

[ ]:
```
def plot_accuracy_loss():
    # plot the accuracy and loss
    train_loss = history.history['loss']
    val_loss = history.history['val_loss']
    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']

    epochsn = np.arange(1, len(train_loss)+1,1)
    plt.figure(figsize=(12, 4))

    plt.subplot(1,2,1)
    plt.plot(epochsn, acc, 'b', label='Training Accuracy')
    plt.plot(epochsn, val_acc, 'r', label='Validation Accuracy')
    plt.grid(color='gray', linestyle='--')
```

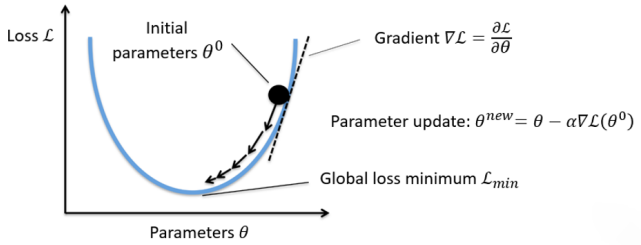
(continues on next page)

(continued from previous page)

```
    plt.legend()
    plt.title('ACCURACY')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')

    plt.subplot(1,2,2)
    plt.plot(epochsn,train_loss, 'b', label='Training Loss')
    plt.plot(epochsn,val_loss, 'r', label='Validation Loss')
    plt.grid(color='gray', linestyle='--')
    plt.legend()
    plt.title('LOSS')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.show()
```

Learning rate = 1e-4, Epochs = 10

In Lecture 15, we explained that most NNs employ a version of the Gradient Descent algorithm for updating the network parameters during training, depicted in the figure below. The learning rate determines the size of the updates at each step, i.e., it controls how fast the network parameters are updated.



Gradient descent algorithm:

1. Randomly initialize the parameters θ^0
2. Compute the gradient of the loss function: $\nabla\mathcal{L}(\theta^0)$
3. Update the parameters: $\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$
   - α is learning rate
4. Go to step 2 and repeat

Figure: Gradient descent algorithm.

If the learning rate is too small, the algorithm will take many epochs to converge, and it may even get stuck into a local minima. If the learning rate is too high, the algorithm may jump over the best solutions and may not be able to converge to good solutions.

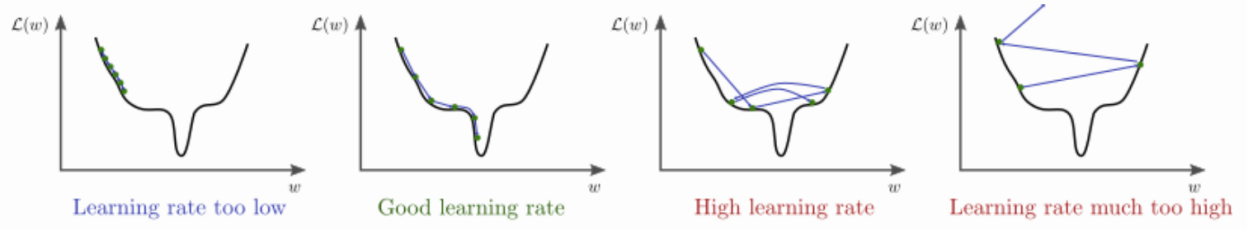


Figure: Impact of the learning rate. Source: https://www.bdhammel.com/learning-rates/

In the previous lectures, we used the following code to compile the models, which uses the Adam optimizer, but we didn't specify the learning rate of the optimizer. For the implementation of Adam in Keras, the default learning rate is 1e-3.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

If we would like to use another value for the learning rate, then we will need to import the Adam optimizer, and compile the model with:

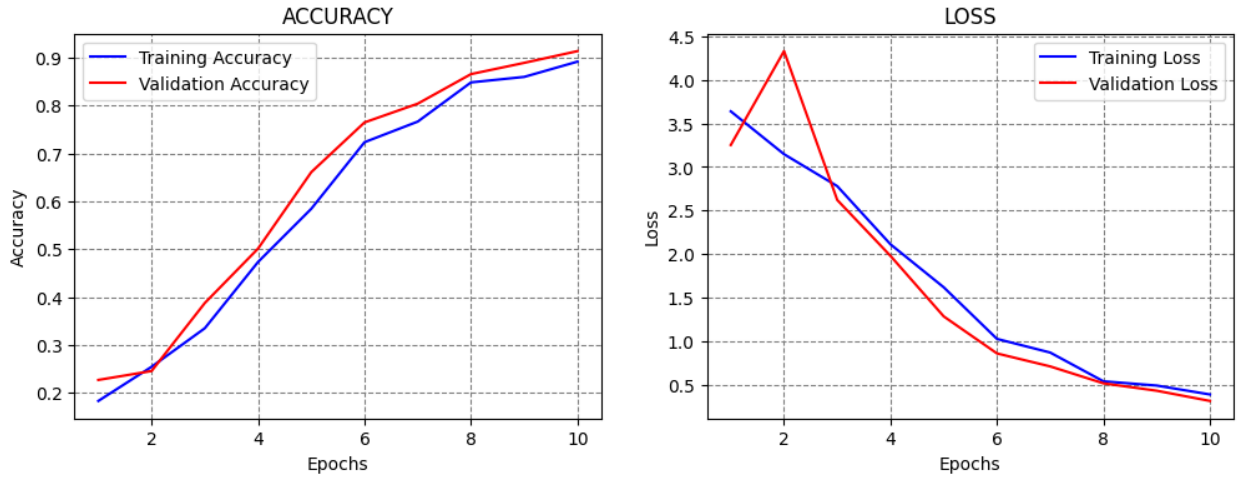
```
from keras.optimizers import Adam
model.compile(optimizer = Adam(learning_rate=VALUE),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

The following code trains a model for 10 epochs with a learning rate of `1e-4 = 0.0001`. We have selected this learning rate because we know that it works well for this combination of model and data.

We can see that the model achieved close to 90% accuracy on the test set, and the training took about 1 minute. From the plots of the accuracy and loss curves, we can tell that 10 epochs are not sufficient for training the model, because at the end of the 10th epoch, the accuracy was still increasing and the loss was decreasing.

[ ]:
```
LEARNING_RATE = 1e-4
EPOCHS_NUM = 10

# create a model
model = Network()

# compile the model
model.compile(optimizer = Adam(learning_rate=LEARNING_RATE), loss='sparse_categorical_
↪crossentropy', metrics=['accuracy'])

# fit model
t = now()
history = model.fit(imgs_train, labels_train, batch_size=32, epochs=EPOCHS_NUM,
                    validation_data=(imgs_val, labels_val), verbose=0)
print('\nTraining time: %s' % (now() - t))

# evaluate on test data
evals_test = model.evaluate(imgs_test, labels_test)
print("Classification Accuracy: ", 100*evals_test[1])

# plot the accuracy and loss
plot_accuracy_loss()
```

```
Training time: 0:00:44.932419
33/33 [==============================] - 1s 21ms/step - loss: 0.3862 - accuracy: 0.8990
Classification Accuracy:  89.89514112472534
```



Learning rate = 1e-4, Epochs = 30

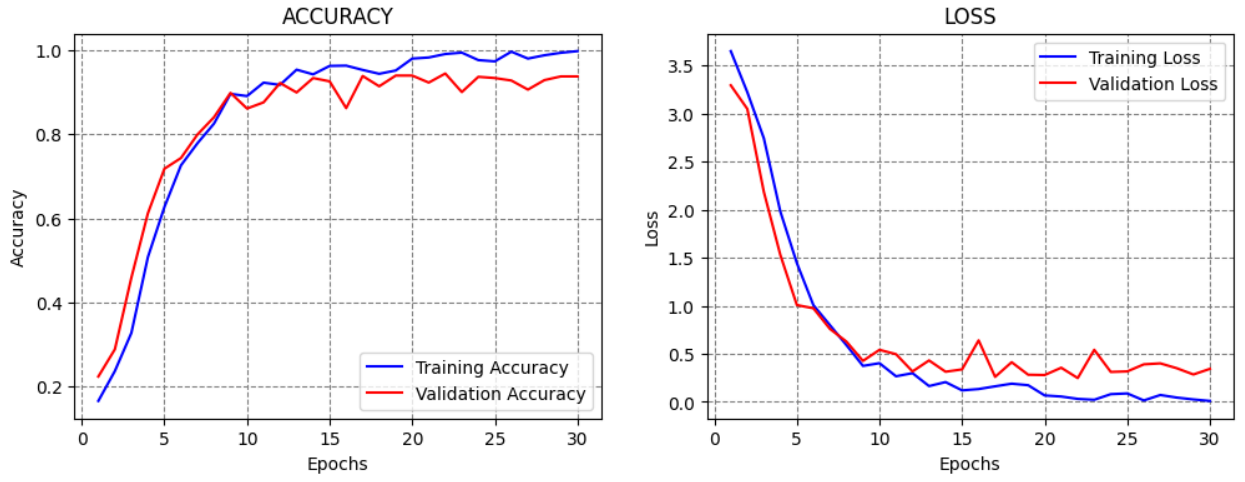
Let's train the model for 30 epochs using the same learning rate to see if this number of epochs would be sufficient.

From the results we can see that the model achieved 94% accuracy, and the training took about 2 minutes.

Based on the learning curves, at epoch 30 the validation accuracy and loss were converging to a plateau level, and it was unclear if training the model for more than 30 epochs would improve the performance. An alternative is to use Early Stopping callback, so that the training is stopped automatically (e.g., when the validation loss stops decreasing). Such case is shown in a subsequent section.

[ ]:
```
LEARNING_RATE = 1e-4
EPOCHS_NUM = 30

model = Network()
model.compile(optimizer = Adam(learning_rate=LEARNING_RATE), loss='sparse_categorical_
↪crossentropy', metrics=['accuracy'])

# fit model
t = now()
history = model.fit(imgs_train, labels_train, batch_size=32, epochs=EPOCHS_NUM,
                    validation_data=(imgs_val, labels_val), verbose=0)
print('\nTraining time: %s' % (now() - t))

# Evaluate on test data
evals_test = model.evaluate(imgs_test, labels_test)
print("Classification Accuracy: ", 100*evals_test[1])

# plot the accuracy and loss
plot_accuracy_loss()
```

```
Training time: 0:01:49.333017
33/33 [==============================] - 0s 12ms/step - loss: 0.3438 - accuracy: 0.9428
Classification Accuracy:  94.28026676177979
```



Learning rate = 1e-3, Epochs = 20

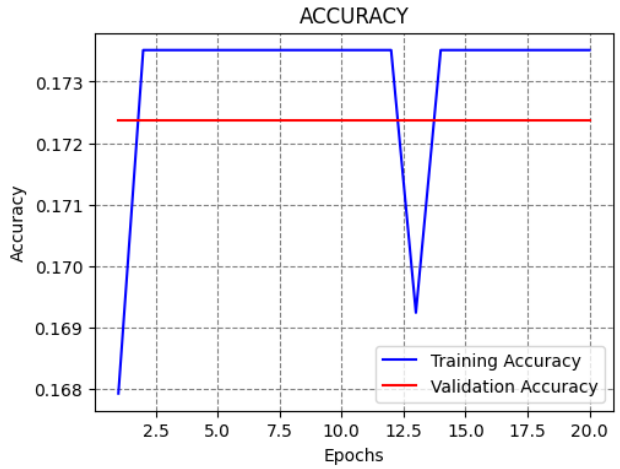
Next, let's try to train the model using different learning rates, for instance, by increasing the learning rate to `1e-3 = 0.001`.

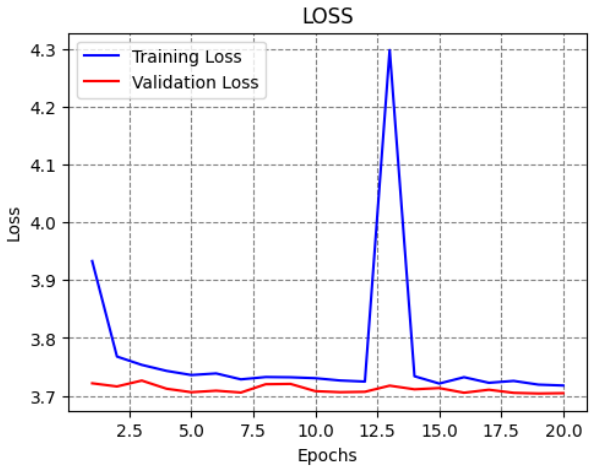
Increasing the learning rate will cause the model to apply larger values for the update of the parameters during the training. We can expect that the training will converge faster, and we can use a smaller number of epochs.

However, large learning rates can cause the model to update the parameters too fast, as in this case. As we can see, the model achieved only 16.87% accuracy, and the accuracy curves did not improve after that level.

[ ]:
```
LEARNING_RATE = 1e-3
EPOCHS_NUM = 20

model = Network()
model.compile(optimizer = Adam(learning_rate=LEARNING_RATE), loss='sparse_categorical_
↪crossentropy', metrics=['accuracy'])

# fit model
t = now()
history = model.fit(imgs_train, labels_train, batch_size=32, epochs=EPOCHS_NUM,
                    validation_data=(imgs_val, labels_val), verbose=0)
print('\nTraining time: %s' % (now() - t))

# Evaluate on test data
evals_test = model.evaluate(imgs_test, labels_test)
print("Classification Accuracy: ", 100*evals_test[1])

# plot the accuracy and loss
plot_accuracy_loss()
```

```
Training time: 0:01:13.950914
33/33 [==============================] - 0s 11ms/step - loss: 3.7229 - accuracy: 0.1687
Classification Accuracy:  16.873212158679962
```



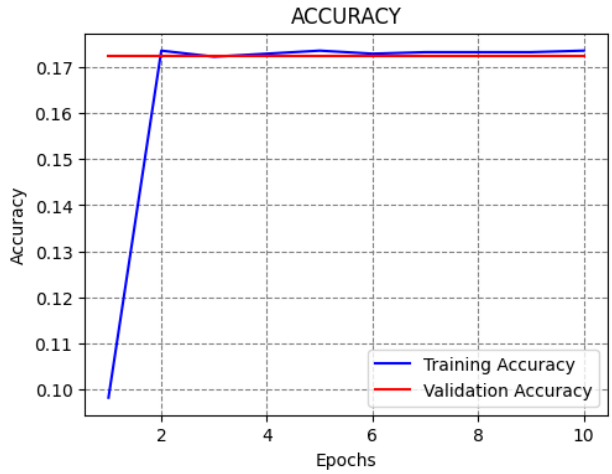


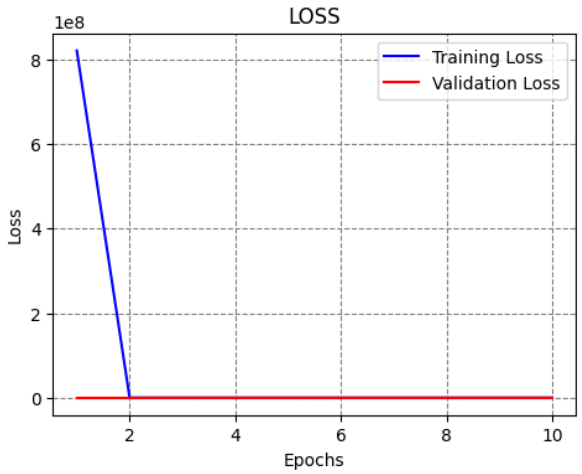
Learning rate = 1e-2, Epochs = 10

If we increase the learning rate even further to `0.01`, we can expect that training will fail, since we saw that even a learning rate of `0.001` was too high. Based on the learning curves, we can tell that the learning is too fast and too aggressive.

[ ]:
```
LEARNING_RATE = 1e-2
EPOCHS_NUM = 10

model = Network()
model.compile(optimizer = Adam(learning_rate=LEARNING_RATE), loss='sparse_categorical_
↪crossentropy', metrics=['accuracy'])

# fit model
t = now()
history = model.fit(imgs_train, labels_train, batch_size=32, epochs=EPOCHS_NUM,
                    validation_data=(imgs_val, labels_val), verbose=0)
print('\nTraining time: %s' % (now() - t))

# Evaluate on test data
evals_test = model.evaluate(imgs_test, labels_test)
print("Classification Accuracy: ", 100*evals_test[1])

# plot the accuracy and loss
plot_accuracy_loss()
```

```
Training time: 0:00:39.685677
33/33 [==============================] - 0s 12ms/step - loss: 3.7224 - accuracy: 0.1687
Classification Accuracy:  16.873212158679962
```





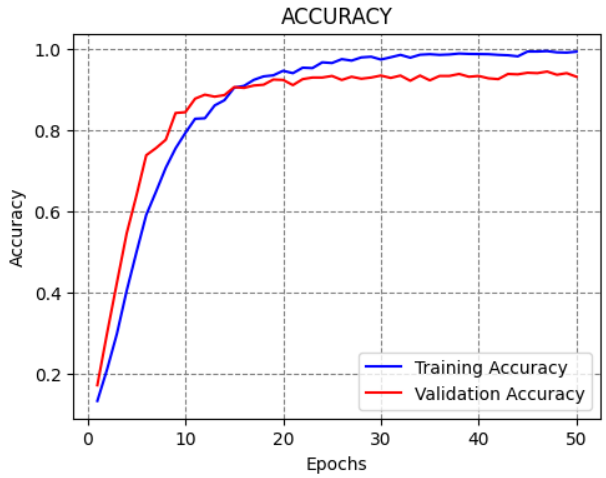
Learning rate = 1e-5, Epochs = 50

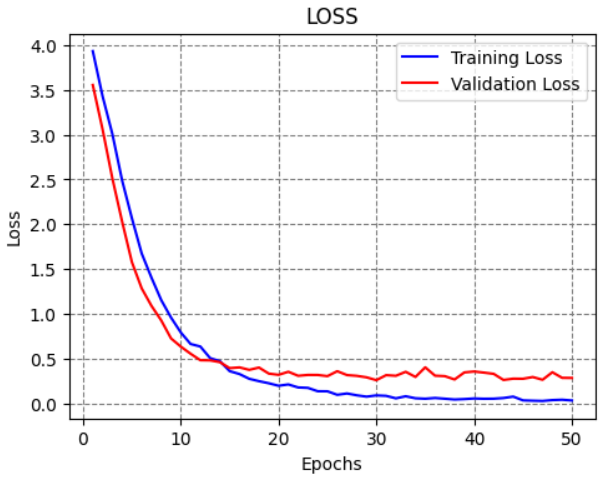
Let's try the opposite case, and reduce the learning rate to `1e-5 = 0.00001`. Smaller learning rates produce smaller updates of the model parameters, and slower learning. This may avoid the problems of learning too fast when large learning rates are used.

This model achieved similar accuracy as with the `1e-4` learning rate. Also, the learning curves look good, because the accuracy and loss gradually change, and the validation curves follow the training curves.

[ ]:
```
LEARNING_RATE = 1e-5
EPOCHS_NUM = 50

model = Network()
model.compile(optimizer = Adam(learning_rate=LEARNING_RATE), loss='sparse_categorical_
↪crossentropy', metrics=['accuracy'])

# fit model
t = now()
history = model.fit(imgs_train, labels_train, batch_size=32, epochs=EPOCHS_NUM,
                    validation_data=(imgs_val, labels_val), verbose=0)
print('\nTraining time: %s' % (now() - t))

# Evaluate on test data
evals_test = model.evaluate(imgs_test, labels_test)
print("Classification Accuracy: ", 100*evals_test[1])

# plot the accuracy and loss
plot_accuracy_loss()
```

```
Training time: 0:02:58.985777
33/33 [==============================] - 0s 12ms/step - loss: 0.2841 - accuracy: 0.9409
Classification Accuracy:  94.08960938453674
```





Learning rate = 1e-6, Epochs = 50

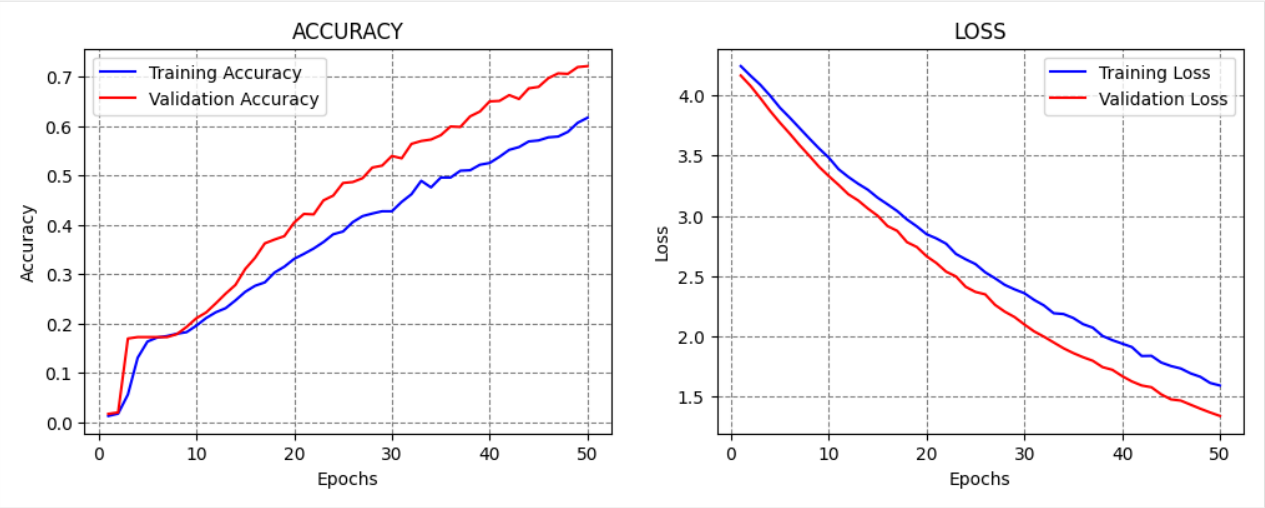
Next, let's reduce the learning rate even further to `1e-6`.

Although smaller learning rates avoid training failure when we use overly large learning rates, using very small learning rates does not necessarily lead to improved performance, since the learning can be too slow.

Based on the learning curves, we can tell that at the end of epoch 50 the model parameters were gradually being updated, and that with this learning rate we would need to train the model for at least 200 or 300 epochs to reach convergence, or maybe even more.

[ ]:
```
LEARNING_RATE = 1e-6
EPOCHS_NUM = 50

model = Network()
model.compile(optimizer = Adam(learning_rate=LEARNING_RATE), loss='sparse_categorical_
↪crossentropy', metrics=['accuracy'])

# fit model
t = now()
history = model.fit(imgs_train, labels_train, batch_size=32, epochs=EPOCHS_NUM,
                    validation_data=(imgs_val, labels_val), verbose=0)
print('\nTraining time: %s' % (now() - t))

# Evaluate on test data
evals_test = model.evaluate(imgs_test, labels_test)
print("Classification Accuracy: ", 100*evals_test[1])

# plot the accuracy and loss
plot_accuracy_loss()
```

```
Training time: 0:02:58.017459
33/33 [==============================] - 0s 12ms/step - loss: 1.3921 - accuracy: 0.6969
Classification Accuracy:  69.68541741371155
```



17.2.1 Learning Rate Finder

There are several tools for estimating the learning rate, such as `LRFinder` in Keras. This function changes the learning rate in a range of values, typically starting with a very small learning rate and increasing it to a high learning rate. The model is trained and evaluated only for a few epochs using different learning rates. Based on a plot of the loss for different learning rates, this function can help to find a suitable learning rate, that can afterward be used to fully train the model.

For the task at hand, the plot of loss versus learning rate is shown below. The best learning rate is the one when the loss is reducing the fastest, and has the largest slope. From the graph, this value is about 10^{-4}.

Although such tools can identify a range of suitable values for the learning rate, they should be used with caution, and the users should still run candidate models in the suggested range to fully evaluate the model performance.

[ ]:
```
!git clone https://github.com/WittmannF/LRFinder.git
```

```
Cloning into 'LRFinder'...
remote: Enumerating objects: 71, done.
remote: Total 71 (delta 0), reused 0 (delta 0), pack-reused 71
Receiving objects: 100% (71/71), 447.02 KiB | 12.08 MiB/s, done.
Resolving deltas: 100% (24/24), done.
```

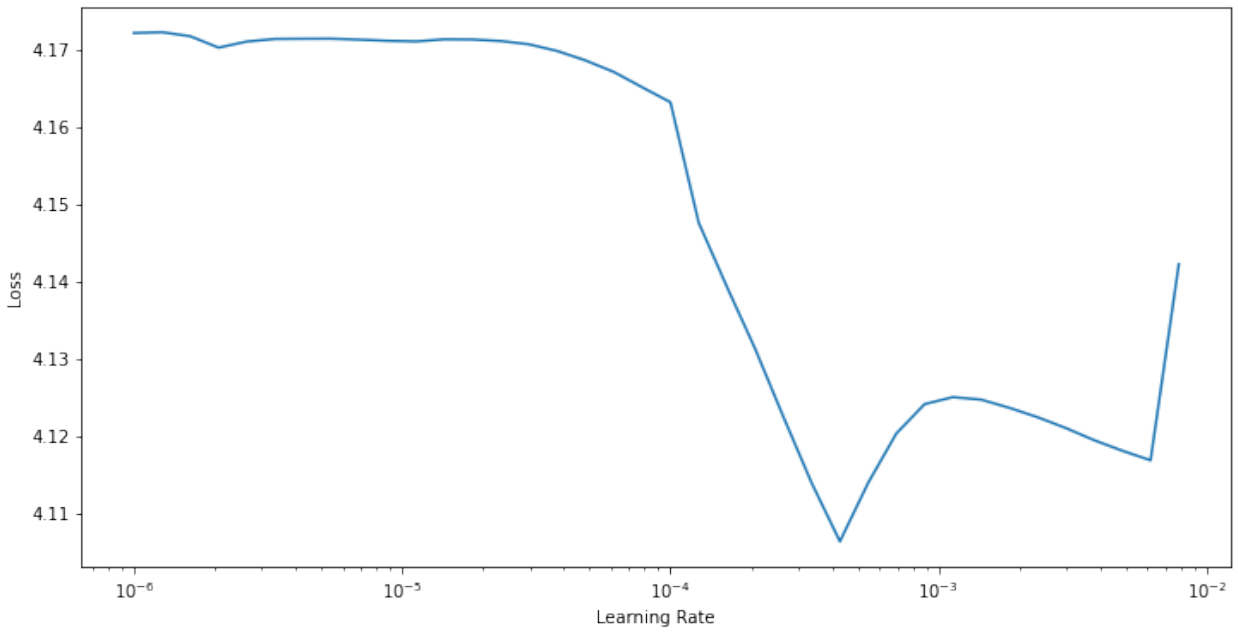
[ ]:
```
from LRFinder.keras_callback import LRFinder
```

[ ]:
```
model = Network()
model.compile(optimizer='Adam', loss='sparse_categorical_crossentropy', metrics=[
↪'accuracy'])

# Perform the Learning Rate Range Test
lr_finder = LRFinder(min_lr=1e-6, max_lr=1e-2)

model.fit(imgs_train, labels_train, batch_size=32, callbacks=[lr_finder], epochs=2)
```

```
Epoch 1/2
 6/96 [>...] - ETA: 11s - loss: 4.1732 - accuracy: 0.0000e+00
```

```
WARNING:tensorflow:Callback method `on_train_batch_end` is slow compared to the batch␣
↪time (batch time: 0.0407s vs `on_train_batch_end` time: 0.0935s). Check your callbacks.
```

```
96/96 [==============================] - 36s 122ms/step - loss: 4.1457 - accuracy: 0.0214
Epoch 2/2
96/96 [==============================] - 12s 120ms/step - loss: 3660.4465 - accuracy: 0.
↪0796
```




```
<keras.callbacks.History at 0x7f5c50298650>
```

7.17.3 17.3 Callbacks

Callbacks in programming languages are functions that allow for conditional processing within another function. I.e., they allow to perform some operations during the execution of another function, based on some conditions.

In ML, callbacks are used to monitor the model performance at various stages of training and take certain actions (e.g., at the start or end of an epoch, before or after processing a single batch, etc.). Such actions can include saving the model to the disk after a certain number of epochs, obtaining a view of the internal states of the model and relevant statistics during training, writing logs after every batch of training data to monitor performance metrics, etc. Most ML libraries provide a callback class which allows users to create custom callbacks.

So far we worked with the `history` callback in Keras, which stores the values of the loss and the metric at each epoch, and its atribute `history.history` allows to plot the learning curves of a model. Also, in the lecture on ConvNets we explained how the `EarlyStopping` callback works. The next section provides additional examples of applying callbacks in Keras.

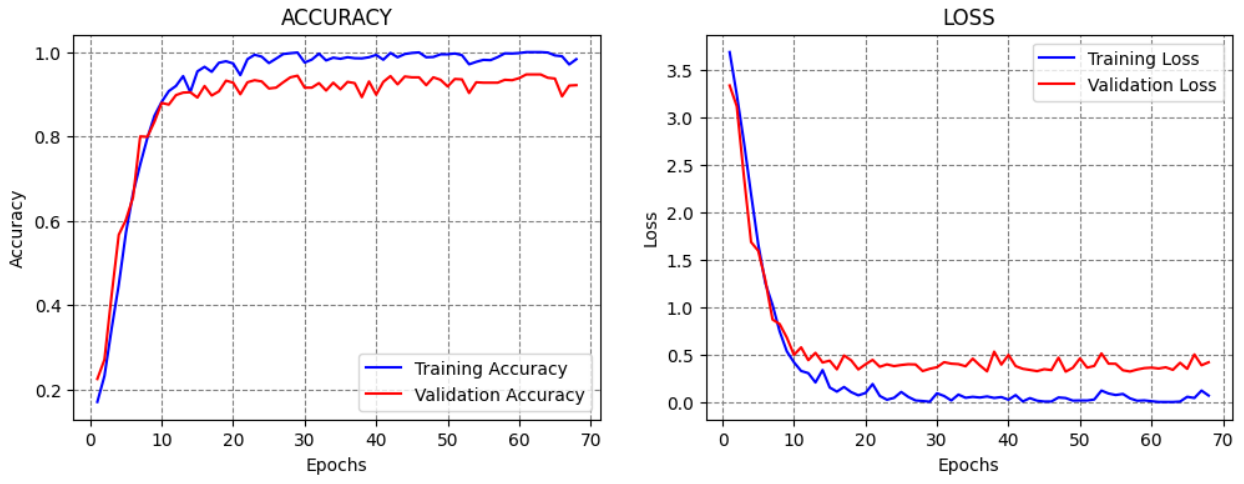
17.3.1 Early Stopping

As we know, using **Early Stopping** callback is often beneficial, since we don't need to guess the optimal number of epochs to train the model. Instead, the callback will terminate the training when a selected metric is not improving. In the next cell, we specified to stop the training when the validation loss does not improve for 20 epochs (`patience` argument). We set the `EPOCH_NUM` argument to 1000, although we know that the model will terminate after about 50-60 epochs. Therefore, the number of epochs is not very important when we use this callback, it only needs to be large enough so that the training is not stopped prematurely.

This model achieved 91.13% accuracy. However, from the accuracy curve it seems that the accuracy was higher in the previous epoch, and it just dropped in the last epoch. We will see next how to avoid this by using CheckPoint callback.

[ ]:
```
LEARNING_RATE = 1e-4
EPOCHS_NUM = 1000

model = Network()
model.compile(optimizer=Adam(learning_rate=LEARNING_RATE), loss='sparse_categorical_
↪crossentropy', metrics=['accuracy'])

# fit model
t = now()
callbacks = [EarlyStopping(monitor='val_loss', patience=20)]
history = model.fit(imgs_train, labels_train, batch_size=32, epochs=EPOCHS_NUM,
                    validation_data=(imgs_val, labels_val), verbose=0,␣
↪callbacks=callbacks)
print('\nTraining time: %s' % (now() - t))

# Evaluate on test data
evals_test = model.evaluate(imgs_test, labels_test)
print("Classification Accuracy: ", 100*evals_test[1])

# plot the accuracy and loss
plot_accuracy_loss()
```

```
Training time: 0:04:01.178825
33/33 [==============================] - 0s 12ms/step - loss: 0.4387 - accuracy: 0.9256
Classification Accuracy:  92.56434440612793
```



17.3.2 Save CheckPoint

CheckPoint callback saves a checkpoint of the model (i.e., the model parameters) after every epoch when a monitored metric does not improve. We set the metrics to be the validation loss, and the values of the model parameters will be saved at the specified `filepath`. This can be useful if training the model takes hours, where if something goes wrong, we can just resume the training from a checkpoint.

We choose to set `verbose=1` for the callback, to output the epochs at which a checkpoint is saved.

At the beginning of the training, the checkpoint will be saved after every epoch, and after the model reaches a plateau, a new checkpoint will be saved only when there is an improvement in the performance, by overwriting the latest checkpoint.

[ ]:
```
LEARNING_RATE = 1e-4
EPOCHS_NUM = 50

model = Network()
model.compile(optimizer=Adam(learning_rate=LEARNING_RATE), loss='sparse_categorical_
↪crossentropy', metrics=['accuracy'])

# fit model
t = now()
callbacks = ModelCheckpoint(filepath='sample_data/model_celeb.h5', monitor='val_loss',␣
↪mode='min',
                            save_weights_only=True, save_best_only=True, verbose=1)
history = model.fit(imgs_train, labels_train, batch_size=32, epochs=EPOCHS_NUM,
                    validation_data=(imgs_val, labels_val), verbose=0,␣
↪callbacks=[callbacks])
print('Training time: %s' % (now() - t))

# Evaluate on test data
evals_test = model.evaluate(imgs_test, labels_test)
print("Classification Accuracy: ", 100*evals_test[1])

# plot the accuracy and loss
plot_accuracy_loss()
```

```
Epoch 1: val_loss improved from inf to 3.32760, saving model to sample_data/model_celeb.
↪h5

Epoch 2: val_loss improved from 3.32760 to 3.08174, saving model to sample_data/model_
↪celeb.h5

Epoch 3: val_loss improved from 3.08174 to 2.40263, saving model to sample_data/model_
↪celeb.h5

Epoch 4: val_loss improved from 2.40263 to 1.87908, saving model to sample_data/model_
↪celeb.h5

Epoch 5: val_loss improved from 1.87908 to 1.77787, saving model to sample_data/model_
↪celeb.h5

Epoch 6: val_loss improved from 1.77787 to 1.03076, saving model to sample_data/model_
↪celeb.h5
```

(continues on next page)

(continued from previous page)

```
Epoch 7: val_loss improved from 1.03076 to 0.86254, saving model to sample_data/model_
↪celeb.h5

Epoch 8: val_loss improved from 0.86254 to 0.83952, saving model to sample_data/model_
↪celeb.h5

Epoch 9: val_loss improved from 0.83952 to 0.76099, saving model to sample_data/model_
↪celeb.h5

Epoch 10: val_loss improved from 0.76099 to 0.45982, saving model to sample_data/model_
↪celeb.h5

Epoch 11: val_loss improved from 0.45982 to 0.39744, saving model to sample_data/model_
↪celeb.h5

Epoch 12: val_loss improved from 0.39744 to 0.36517, saving model to sample_data/model_
↪celeb.h5

Epoch 13: val_loss improved from 0.36517 to 0.35960, saving model to sample_data/model_
↪celeb.h5

Epoch 14: val_loss did not improve from 0.35960

Epoch 15: val_loss did not improve from 0.35960

Epoch 16: val_loss did not improve from 0.35960

Epoch 17: val_loss improved from 0.35960 to 0.33312, saving model to sample_data/model_
↪celeb.h5

Epoch 18: val_loss did not improve from 0.33312

Epoch 19: val_loss did not improve from 0.33312

Epoch 20: val_loss improved from 0.33312 to 0.31346, saving model to sample_data/model_
↪celeb.h5

Epoch 21: val_loss did not improve from 0.31346

Epoch 22: val_loss did not improve from 0.31346

Epoch 23: val_loss did not improve from 0.31346

Epoch 24: val_loss improved from 0.31346 to 0.30755, saving model to sample_data/model_
↪celeb.h5

Epoch 25: val_loss did not improve from 0.30755

Epoch 26: val_loss did not improve from 0.30755

Epoch 27: val_loss did not improve from 0.30755
```

(continues on next page)

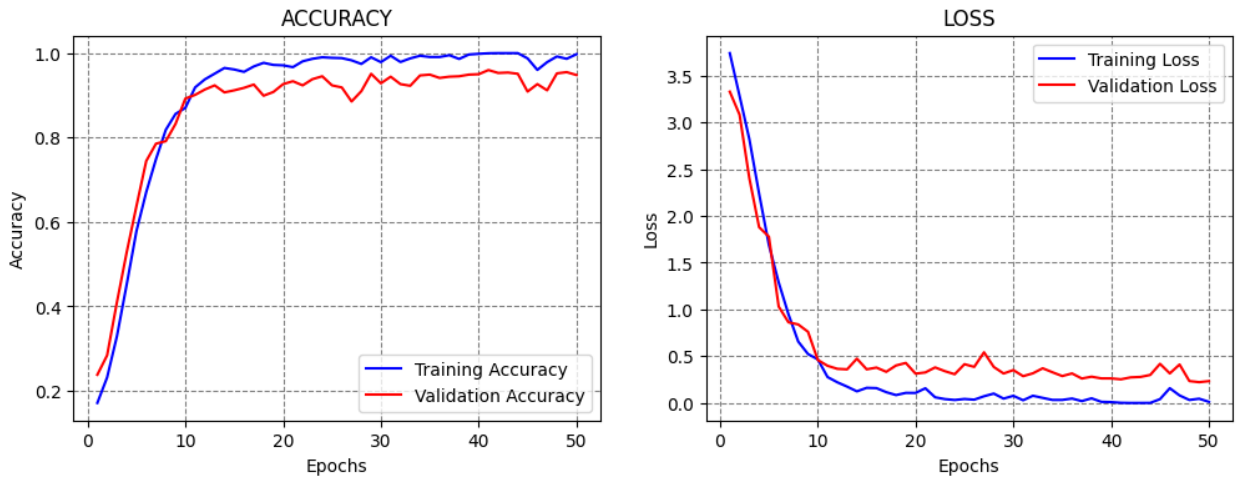
(continued from previous page)

```
Epoch 28: val_loss did not improve from 0.30755

Epoch 29: val_loss did not improve from 0.30755

Epoch 30: val_loss did not improve from 0.30755

Epoch 31: val_loss improved from 0.30755 to 0.28651, saving model to sample_data/model_
↪celeb.h5

Epoch 32: val_loss did not improve from 0.28651

Epoch 33: val_loss did not improve from 0.28651

Epoch 34: val_loss did not improve from 0.28651

Epoch 35: val_loss did not improve from 0.28651

Epoch 36: val_loss did not improve from 0.28651

Epoch 37: val_loss improved from 0.28651 to 0.26158, saving model to sample_data/model_
↪celeb.h5

Epoch 38: val_loss did not improve from 0.26158

Epoch 39: val_loss did not improve from 0.26158

Epoch 40: val_loss did not improve from 0.26158

Epoch 41: val_loss improved from 0.26158 to 0.25302, saving model to sample_data/model_
↪celeb.h5

Epoch 42: val_loss did not improve from 0.25302

Epoch 43: val_loss did not improve from 0.25302

Epoch 44: val_loss did not improve from 0.25302

Epoch 45: val_loss did not improve from 0.25302

Epoch 46: val_loss did not improve from 0.25302

Epoch 47: val_loss did not improve from 0.25302

Epoch 48: val_loss improved from 0.25302 to 0.23428, saving model to sample_data/model_
↪celeb.h5

Epoch 49: val_loss improved from 0.23428 to 0.22223, saving model to sample_data/model_
↪celeb.h5

Epoch 50: val_loss did not improve from 0.22223
Training time: 0:03:01.450305
```

(continues on next page)

(continued from previous page)

```
33/33 [==============================] - 0s 11ms/step - loss: 0.2903 - accuracy: 0.9561
Classification Accuracy:  95.61486840248108
```



17.3.3 Reduce Learning Rate On Plateau

ReduceLROnPlateau stands for Reduce Learning Rate on Plateau. It is another very useful callback, since prior works have reported that training models generally benefits from using larger learning rate at the beginning of the training, and gradually reducing the learning rate when the training does not improve.

This is exactly what this callback does. In the next call, the learning rate is initially set to `1e-4=0.0001`. ReduceLROnPlateau has a patience of 10 epochs, factor of 0.1, and minimum learning rate of `1e-6`. This means that when the monitored metric (in this case, the validation loss) does not reduce for 10 epochs, the learning rate will be multiplied by the `factor` and become `1e-5`. When the model stops improving again, the learning rate will be again multiplied by the factor and become `1e-6`. Since this is the minimum value for the learning rate, we will combine this callback with Early Stopping to terminate the training. Note that the patience value for Early Stopping was set longer than the patience for ReduceLROnPlateau.

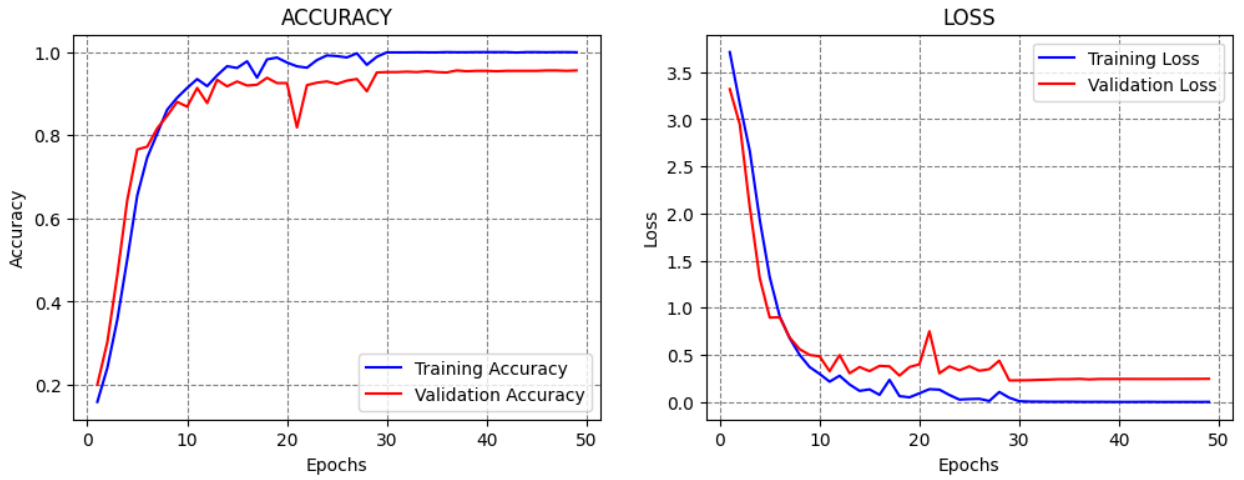
[ ]:
```
LEARNING_RATE = 1e-4
EPOCHS_NUM = 1000

model = Network()
model.compile(optimizer=Adam(learning_rate = LEARNING_RATE), loss='sparse_categorical_
↪crossentropy', metrics=['accuracy'])

# fit model
t = now()
callbacks = [EarlyStopping(monitor='val_loss', patience = 20),
             ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=10, min_lr=1e-6,␣
↪verbose=1)]
history = model.fit(imgs_train, labels_train, batch_size=32, epochs=EPOCHS_NUM,
                    validation_data=(imgs_val, labels_val), verbose=0,␣
↪callbacks=callbacks)
print('\nTraining time: %s' % (now() - t))

# Evaluate on test data
evals_test = model.evaluate(imgs_test, labels_test)
```

(continues on next page)

(continued from previous page)

```
print("Classification Accuracy: ", 100*evals_test[1])


# plot the accuracy and loss
plot_accuracy_loss()
```

```
Epoch 28: ReduceLROnPlateau reducing learning rate to 9.999999747378752e-06.

Epoch 39: ReduceLROnPlateau reducing learning rate to 1e-06.

Training time: 0:02:54.702996
33/33 [==============================] - 0s 12ms/step - loss: 0.3021 - accuracy: 0.9466
Classification Accuracy:  94.66158151626587
```



17.3.4 Learning Rate Scheduler

Learning Rate Scheduler allows to define custom schedulers for adjusting the learning rate during training. Popular learning rate schedules include:

- Time-based decay
- Step decay
- Exponential decay

Time-based Decay

This scheduler decreases the learning rate in each epoch by a given fixed amount. An example is shown in the next figure, where a model is trained for 100 epochs, and the learning rate is gradually reduced from 0.01 in the first epoch to 0.006 in the last epoch.

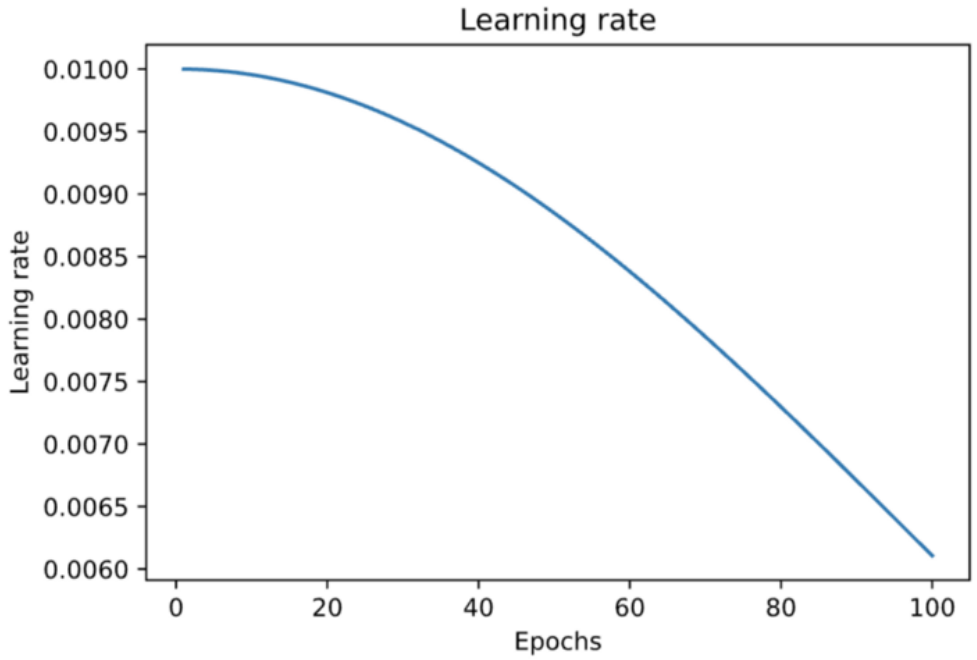
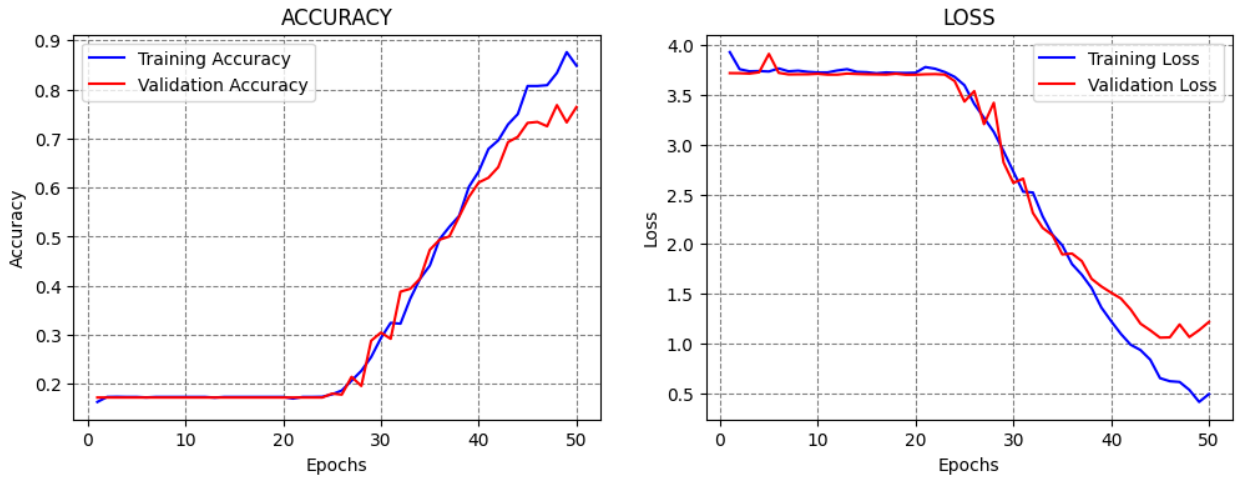


Figure: Time-based decay.

The implementation is shown below, where the Learning Rate Scheduler callback accepts a function which defines the schedule for the learning rate. The function `lr_time_based_decay` applies the `decay` amount at each epoch, where `lr` is the learning rate from the previous epoch. The value of the decay is usually set as the quotient of the initial learning rate and the number of epochs.

[ ]:
```
INITIAL_LEARNING_RATE = 1e-4
EPOCHS_NUM = 50
decay = INITIAL_LEARNING_RATE / EPOCHS_NUM

def lr_time_based_decay(epoch, lr):
    return lr * 1 / (1 + decay * epoch)

model = Network()
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=[
↪'accuracy'])

# fit model
t = now()
callbacks = [LearningRateScheduler(lr_time_based_decay, verbose=0)]
history = model.fit(imgs_train, labels_train, batch_size=32, epochs=EPOCHS_NUM,
                    validation_data=(imgs_val, labels_val), verbose=0,␣
↪callbacks=callbacks)
print('\nTraining time: %s' % (now() - t))

# Evaluate on test data
evals_test = model.evaluate(imgs_test, labels_test)
print("Classification Accuracy: ", 100*evals_test[1])

# plot the accuracy and loss
plot_accuracy_loss()
```

```
Training time: 0:02:57.472491
33/33 [==============================] - 0s 12ms/step - loss: 1.1274 - accuracy: 0.7750
Classification Accuracy:  77.50238180160522
```



In this case, it seems that the learning rate was reduced too fast, because the performance decreased.

Step Decay

Step decay scheduler decreases the learning rate for a fixed amount after a number of training epochs. An example is shown in the figure, where the learning rate is reduced by half every 10 epochs.

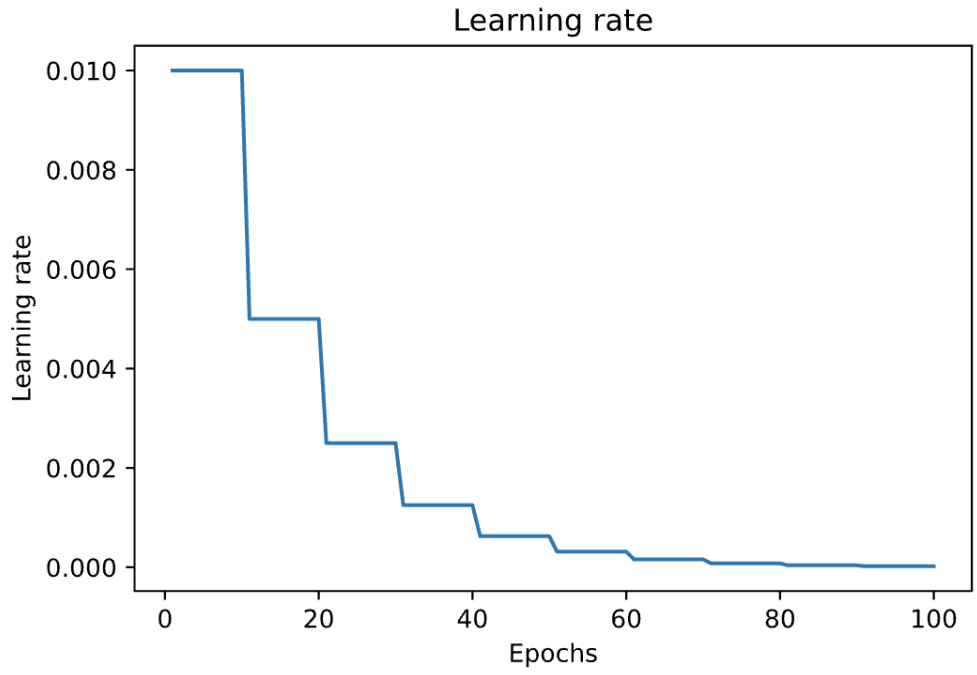
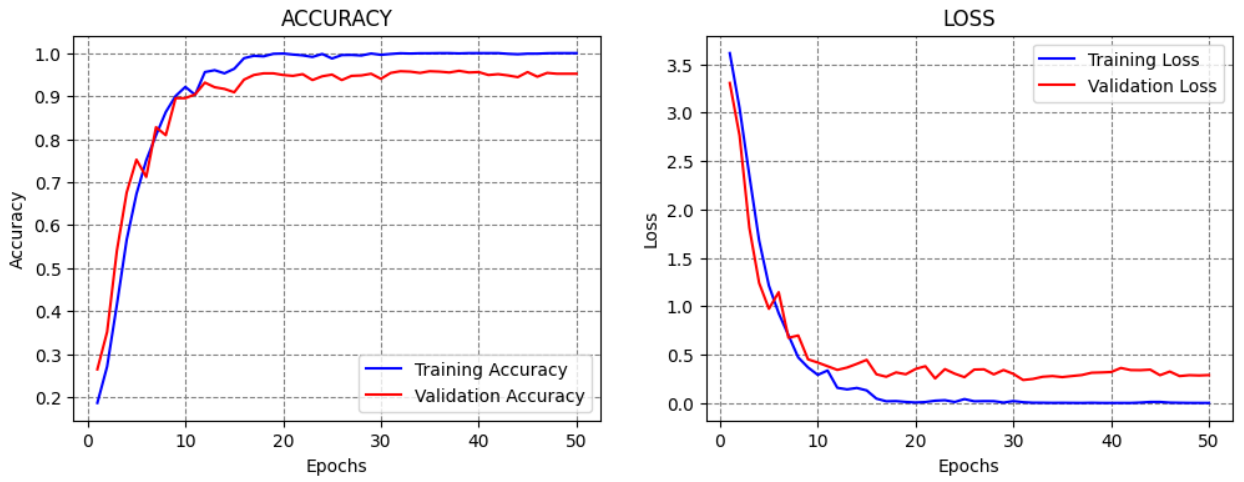


Figure: Step decay.

In the next cell, step decay is applied with the function `lr_step_decay`, where `drop_rate` is the reduced ratio of the initial learning rate at each step, and `epochs_drop` is set to 15 epochs.

[ ]:
```
import math

INITIAL_LEARNING_RATE = 1e-4
EPOCHS_NUM = 50

def lr_step_decay(epoch):
    drop_rate = 0.5
    epochs_drop = 15
    return INITIAL_LEARNING_RATE * math.pow(drop_rate, math.floor(epoch/epochs_drop))

model = Network()
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=[
↪'accuracy'])

# fit model
t = now()
callbacks = [LearningRateScheduler(lr_step_decay, verbose=0)]
history = model.fit(imgs_train, labels_train, batch_size=32, epochs=EPOCHS_NUM,
                    validation_data=(imgs_val, labels_val), verbose=0,␣
↪callbacks=callbacks)
print('\nTraining time: %s' % (now() - t))

# Evaluate on test data
evals_test = model.evaluate(imgs_test, labels_test)
print("Classification Accuracy: ", 100*evals_test[1])

# plot the accuracy and loss
plot_accuracy_loss()
```

```
Training time: 0:02:58.313695
33/33 [==============================] - 0s 12ms/step - loss: 0.3283 - accuracy: 0.9523
Classification Accuracy:  95.233553647995
```



Exponential Decay

The scheduler decreases the learning rate at an exponential rate. In the cell below `k` is the rate of exponential decay.

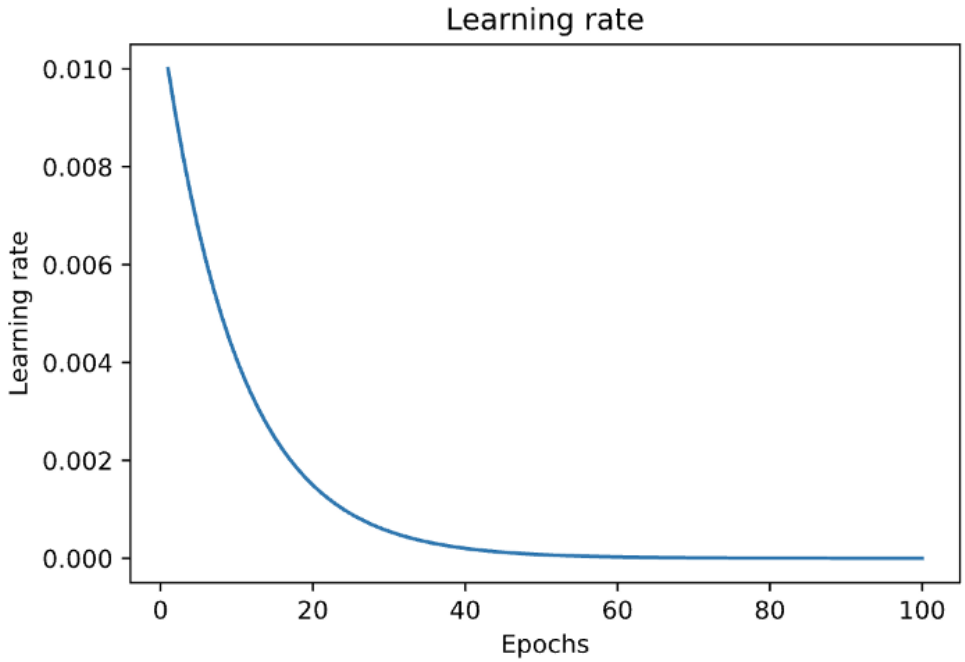
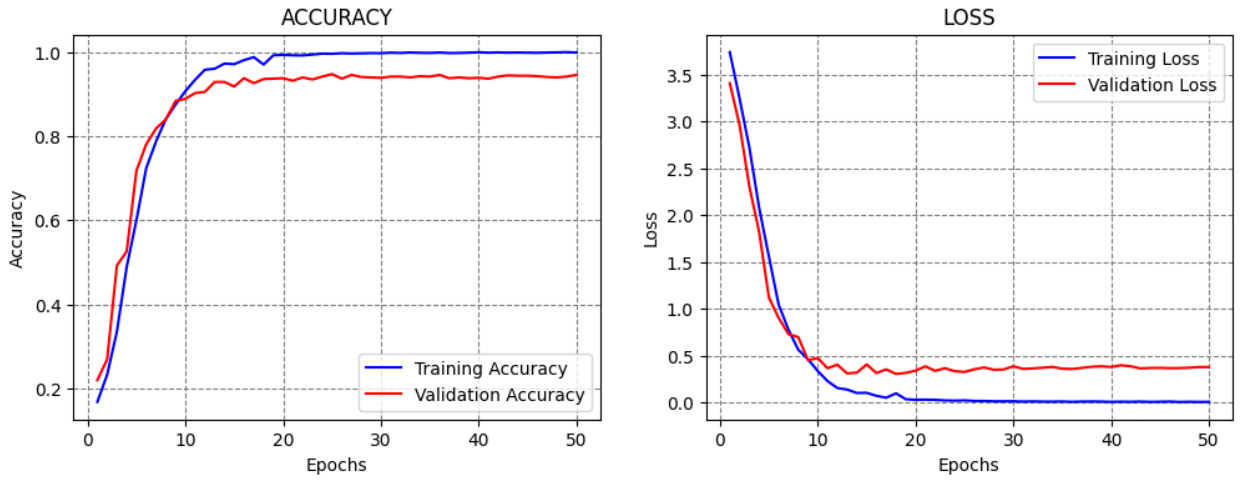


Figure: Exponential decay.

```
[ ]: INITIAL_LEARNING_RATE = 1e-4
EPOCHS_NUM = 50

def lr_exp_decay(epoch):
    k = 0.1
    return INITIAL_LEARNING_RATE * math.exp(-k*epoch)

model = Network()
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=[
↪'accuracy'])

# fit model
t = now()
callbacks = [LearningRateScheduler(lr_exp_decay, verbose=0)]
history = model.fit(imgs_train, labels_train, batch_size=32, epochs=EPOCHS_NUM,
                    validation_data=(imgs_val, labels_val), verbose=0,␣
↪callbacks=callbacks)
print('\nTraining time: %s' % (now() - t))

# Evaluate on test data
evals_test = model.evaluate(imgs_test, labels_test)
print("Classification Accuracy: ", 100*evals_test[1])

# plot the accuracy and loss
plot_accuracy_loss()
```

```
Training time: 0:02:58.286062
33/33 [==============================] - 0s 12ms/step - loss: 0.4014 - accuracy: 0.9447
Classification Accuracy:  94.47092413902283
```



7.17.4 17.4 Grid Search

Applying **Grid Search** over the hyperparameters of neural networks that have long training times can be prohibitively computationally expensive. On the other hand, when working with smaller datasets and models, applying Grid Search for some of the hyperparameters can be an option.

This section presents a simple Grid Search over two hyperparameters in the model: number of neurons in the first Dense layer, and the batch size. We can simply write a function that takes as arguments these two hyperparameters, as in the next cell. Afterward, we can create a for-loop and train and evaluate a model for each combination of the values for the selected hyperparameters.

We examined 3 values for the number of neurons and 3 values for the batch size, resulting in 9 models. The results indicate that the model with 1,024 neurons and 128 batch size achieved the best performance. Consider that training each model took about 10 minutes, and evaluating 9 models took about 1.5 hours. Tuning a large number of hyperparameters over several different values can take long time.

[ ]:
```
def Network(neurons_per_layer, batch_size):

    base_model = vgg16.VGG16(weights='imagenet', include_top=False, input_shape=(image_
↪size, image_size, 3))

    # Add a global spatial average pooling layer
    x = base_model.output
    x = GlobalAveragePooling2D()(x)

    # Add dense layers
    x = Dense(neurons_per_layer, activation='relu')(x)
    x = Dropout(0.25)(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(0.25)(x)
    # Add a softmax layer
    predictions = Dense(62, activation='softmax')(x)
```

(continues on next page)

(continued from previous page)

```
    # The model
    model = Model(inputs=base_model.input, outputs=predictions)

    # Compile
    model.compile(optimizer=Adam(learning_rate=1e-4), loss = 'sparse_categorical_
↪crossentropy', metrics = ['accuracy'])

    # Fit
    model.fit(imgs_train, labels_train, batch_size=batch_size, epochs=30,
                    validation_data=(imgs_val, labels_val), verbose=0)

    # Evaluate on test data
    _, test_acc = model.evaluate(imgs_test, labels_test)

    return test_acc
```

[ ]:
```
neurons_per_layers = [512, 1024, 2048]
batch_sizes = [32, 64, 128]

for number_neurons in neurons_per_layers:
    for batch_size in batch_sizes:
        acc = Network(number_neurons, batch_size)
        print('\nNumber of neurons', number_neurons,
                '\tBath size', batch_size,
                '\tTest accuracy', acc)
```

```
33/33 [==============================] - 1s 38ms/step - loss: 0.3991 - accuracy: 0.9218

Number of neurons 512   Bath size 32    Test accuracy 0.9218302965164185
33/33 [==============================] - 1s 38ms/step - loss: 0.4094 - accuracy: 0.9171

Number of neurons 512   Bath size 64    Test accuracy 0.9170638918876648
33/33 [==============================] - 1s 38ms/step - loss: 0.3639 - accuracy: 0.9399

Number of neurons 512   Bath size 128   Test accuracy 0.9399427771568298
33/33 [==============================] - 1s 38ms/step - loss: 0.3111 - accuracy: 0.9352

Number of neurons 1024  Bath size 32    Test accuracy 0.9351763725280762
33/33 [==============================] - 1s 38ms/step - loss: 0.3525 - accuracy: 0.9342

Number of neurons 1024  Bath size 64    Test accuracy 0.9342230558395386
33/33 [==============================] - 1s 38ms/step - loss: 0.3102 - accuracy: 0.9466

Number of neurons 1024  Bath size 128   Test accuracy 0.9466158151626587
33/33 [==============================] - 1s 38ms/step - loss: 0.5164 - accuracy: 0.9142

Number of neurons 2048  Bath size 32    Test accuracy 0.9142040014266968
33/33 [==============================] - 1s 38ms/step - loss: 0.4070 - accuracy: 0.9066

Number of neurons 2048  Bath size 64    Test accuracy 0.9065777063369751
33/33 [==============================] - 1s 38ms/step - loss: 0.3813 - accuracy: 0.9314
```

(continues on next page)

(continued from previous page)

```
Number of neurons 2048  Bath size 128   Test accuracy 0.9313632249832153
```

7.17.5 17.5 Keras Tuner

There are several libraries developed for tuning the hyperparameters of neural networks. One is the **Keras Tuner** for tuning Keras models.

The Keras Tuner is somewhat similar to the Grid Search and Random Search in scikit-learn, and allows to define the search space for the hyperparameters over which the model will be fit, and it returns an optimal set of hyperparameters.

Keras Tuner is not part of the `Keras` package and it needs to be installed and imported.

[ ]:
```
pip install -q -U keras-tuner
```

```
     129.5/129.5 kB 3.2 MB/s eta 0:00:00
     950.8/950.8 kB 9.0 MB/s eta 0:00:00
```

[ ]:
```
import keras_tuner as kt
```

```
Using TensorFlow backend
```

Load Fashion MNIST Dataset

To demonstrate the use of the Keras Tuner we will work with the Fashion MNIST dataset.

[ ]:
```
(img_train, label_train), (img_test, label_test) = keras.datasets.fashion_mnist.load_
↪data()

# Normalize pixel values between 0 and 1
img_train = img_train.astype('float32') / 255.0
img_test = img_test.astype('float32') / 255.0
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-
↪labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-
↪images-idx3-ubyte.gz
26421880/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-
↪labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-
↪images-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step
```

Model Builder

In the cell below, a function called `model_biilder` is created, which performs search over two hyperparameters:

- Number of neurons in the first Dense layer,
- Learning rate.

In the code, `hp` is an instance of the HyperParameterss class provided by Keras Tuner. The line `hp_units = hp.Int('units', min_value=32, max_value=512, step=32)` defines a grid search for the number of neurons in the Dense layer in the range [32, 64, 96, ..., 512].

Next, a grid search for the learning rate is defined in the range `[1e-2, 1e-3, 1e-4]`.

[ ]:
```
from keras.models import Sequential
from keras.layers import Flatten
from keras.losses import SparseCategoricalCrossentropy


def model_builder(hp):
  model = Sequential()
  model.add(Flatten(input_shape=(28, 28)))

  # Tune the number of units in the first Dense layer
  # Choose an optimal value between 32-512
  hp_units = hp.Int('units', min_value=32, max_value=512, step=32)
  model.add(Dense(units=hp_units, activation='relu'))
  model.add(Dense(10))

  # Tune the learning rate for the optimizer
  # Choose an optimal value from 0.01, 0.001, or 0.0001
  hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])

  model.compile(optimizer=Adam(learning_rate=hp_learning_rate),
                loss=SparseCategoricalCrossentropy(from_logits=True),
                metrics=['accuracy'])

  return model
```

Hyperparameter Tuning

The Keras Tuner has four tuning algorithms available:

- RandomSearch Tuner, similar to the Random Grid in scikit-learn performs a random search over a distribution of values for the hyperparameters.
- Hyperband Tuner, trains a large number of models for a few epochs and carries forward only the top-performing half of models to the next round, to converge to a high-performing model.
- BayesianOptimization Tuner, performs BayesianOptimization by creating a probabilistic mapping of the model to the loss function, and iteratively evaluating promising sets of hyperparameters.
- Sklearn Tuner, designed for use with scikit-learn models.

In the next cell, the Hyperband tuner is used, which has as the arguments the model, objective (metric to monitor), maximum number of epochs for each configuration of hyperparameters (training will be stopped for the worst-performing

configurations after this number of epochs), and factor (used to search for top-performing models, e.g., a reduction factor of 3 means that one third of the configurations will be kept for the next iteration, and the rest of the configurations will be eliminated).

[ ]:
```
tuner = kt.Hyperband(model_builder,
                     objective='val_accuracy',
                     max_epochs=10,
                     factor=3)
```

[ ]:
```
tuner.search(img_train, label_train, epochs=50, validation_split=0.2,␣
↪callbacks=[EarlyStopping(monitor='val_loss', patience=5)])
```

```
Trial 30 Complete [00h 00m 41s]
val_accuracy: 0.8767499923706055

Best val_accuracy So Far: 0.8920833468437195
Total elapsed time: 00h 08m 42s
```

[ ]:
```
# Get the optimal hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

print(f"Optimal number of neuron in the Dense layer: {best_hps.get('units')}")
print (f"Optimal learning rate: {best_hps.get('learning_rate')}")
```

```
Optimal number of neuron in the Dense layer: 480
Optimal learning rate: 0.001
```

Train and Evaluate the Model

Next, we will use the optimal hyperparameters from the Keras Tuner to create a model, and afterward we will evaluate the accuracy on the test dataset.

[ ]:
```
# Build the model with the optimal hyperparameters and train it on the data for 50 epochs
model = tuner.hypermodel.build(best_hps)
model.fit(img_train, label_train, epochs=50, validation_split=0.2, verbose=0)

eval_result = model.evaluate(img_test, label_test)
print("[test loss, test accuracy]:", eval_result)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.5827 - accuracy: 0.8874
[test loss, test accuracy]: [0.5827264189720154, 0.8873999714851379]
```

Other libraries that perform model selection, hyperparameter tuning, and neural architecture search include AutoKeras, auto-sklearn, Auto-PyTorch and Ray Tune for PyTorch, AutoWEKA, and others.

7.17.6 17.6 AutoML

AutoML or Automated Machine Learning refers to tools and libraries that are designed to allow non-ML experts to build Machine Learning systems and solve Data Science tasks without extensive knowledge in these fields.

AutoML systems range from automated *No Code* ML solutions that allow the end-users to just drag-and-drop their data, or *Low Code* systems that automate ML steps with minimal coding efforts, to systems that require coding experience and are designed to increase the efficiency of data scientists by automating hyperparameter tuning and architecture search.

Most large providers of cloud computing and ML services typically provide some form of AutoML services. Examples include GoogleAutoML, Microsoft Azure AutoML, Amazon SageMaker, etc.

In a subsequent lecture on training and deploying a model on the cloud, we will present examples of training ML models using No Code and Low Code modes with Microsoft Azure ML.

7.17.7 References

1. TensorFlow - ML Basics with Keras, Introduction to the Keras Tuner, available at https://www.tensorflow.org/tutorials/keras/keras_tuner#:~:text=The%20Keras%20Tuner%20is%20a,called%20hyperparameter%20tuning%20or%20hypertuning.
2. Keras Learning Rate Finder, available at https://github.com/surmenok/keras_lr_finder.
3. Learning Rate Schedule in Practice: an example with Keras and TensorFlow 2.0, B. Chen, available at https://towardsdatascience.com/learning-rate-schedule-in-practice-an-example-with-keras-and-tensorflow-2-0-2f48b2888a0c.
4. AutoML.org Freiburg-Hannover, AutoML, available at https://www.automl.org/automl/.

BACK TO TOP

7.18 Lecture 18 - Neural Networks with PyTorch

7.18.1 18.1 Introduction to PyTorch

PyTorch is currently one of the most popular deep learning frameworks. It is an open-source library built upon the Torch library, and it was developed by Meta AI (previously Facebook AI). It is now part of the Linux Foundation.

As we learned in Tutorial 9, PyTorch provides tensor operations that can conveniently be performed using CPU or GPU devices. It also provides automatic differentiation operations (auotgrad) with Neural Networks.

In this lecture we will explain how to train Neural Networks with PyTorch. PyTorch has similar functionality to Keras and TensorFlow, as it allows to import neural network layers, offers loss functions and optimizers, etc. It has a slightly lower-level of abstraction in comparison to Keras. On the other hand, there are high-level libraries for PyTorch, such as PyTorch Lightning and fast.ai.

Let's import the required libraries. In the next cell, we imported `torch`, several utility functions, and modules such as `torch.nn` that provides functions and tools for building and training neural networks, and `torchvision` that provides functions and tools for image processing and related computer vision tasks.

[ ]:
```
# import libraries
import numpy as np
import matplotlib.pyplot as plt
import os
import pandas as pd
from PIL import Image

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader, Subset
import torch.optim as optim
# trochvision is part of PyTorch consisting of models and datasets for computer vision
import torchvision
from torchvision import datasets, transforms
from torchvision.datasets import ImageFolder
```

7.18.2 18.2 Loading the Dataset

We will use one of the most common deep learning datasets - MNIST (Modified National Institute of Standards and Technology database). MNIST is a dataset of handwritten digits from 0 to 9, containing 60,000 training images and 10,000 testing images. Each image has 28x28 pixels size.

PyTorch provides access to several datasets, and MNIST can be loaded conveniently by using the `datasets.MNIST` function. In the used arguments, `root` is the directory where the dataset exists, and `transform` can be used to apply data scaling, image resizing, or other transformation operations. Such operations are not required for this dataset.

[ ]:
```
training_data = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=transforms.ToTensor()
)

test_data = datasets.MNIST(
    root="data",
    train=False,
    download=True,
```

(continues on next page)

(continued from previous page)

```
    transform=transforms.ToTensor()
)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to data/MNIST/
↪raw/train-images-idx3-ubyte.gz
```

```
100%|| 9912422/9912422 [00:00<00:00, 90677866.79it/s]
```

```
Extracting data/MNIST/raw/train-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to data/MNIST/
↪raw/train-labels-idx1-ubyte.gz
```

```
100%|| 28881/28881 [00:00<00:00, 27148295.34it/s]
```

```
Extracting data/MNIST/raw/train-labels-idx1-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to data/MNIST/raw/
↪t10k-images-idx3-ubyte.gz
```

```
100%|| 1648877/1648877 [00:00<00:00, 23498662.62it/s]
```

```
Extracting data/MNIST/raw/t10k-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to data/MNIST/raw/
↪t10k-labels-idx1-ubyte.gz
```

```
100%|| 4542/4542 [00:00<00:00, 4444827.06it/s]
```

```
Extracting data/MNIST/raw/t10k-labels-idx1-ubyte.gz to data/MNIST/raw
```

[ ]:
```
print(training_data)
```

```
Dataset MNIST
    Number of datapoints: 60000
    Root location: data
    Split: Train
    StandardTransform
Transform: ToTensor()
```

[ ]:
```
print(test_data)
```

```
Dataset MNIST
    Number of datapoints: 10000
    Root location: data
    Split: Test
    StandardTransform
Transform: ToTensor()
```

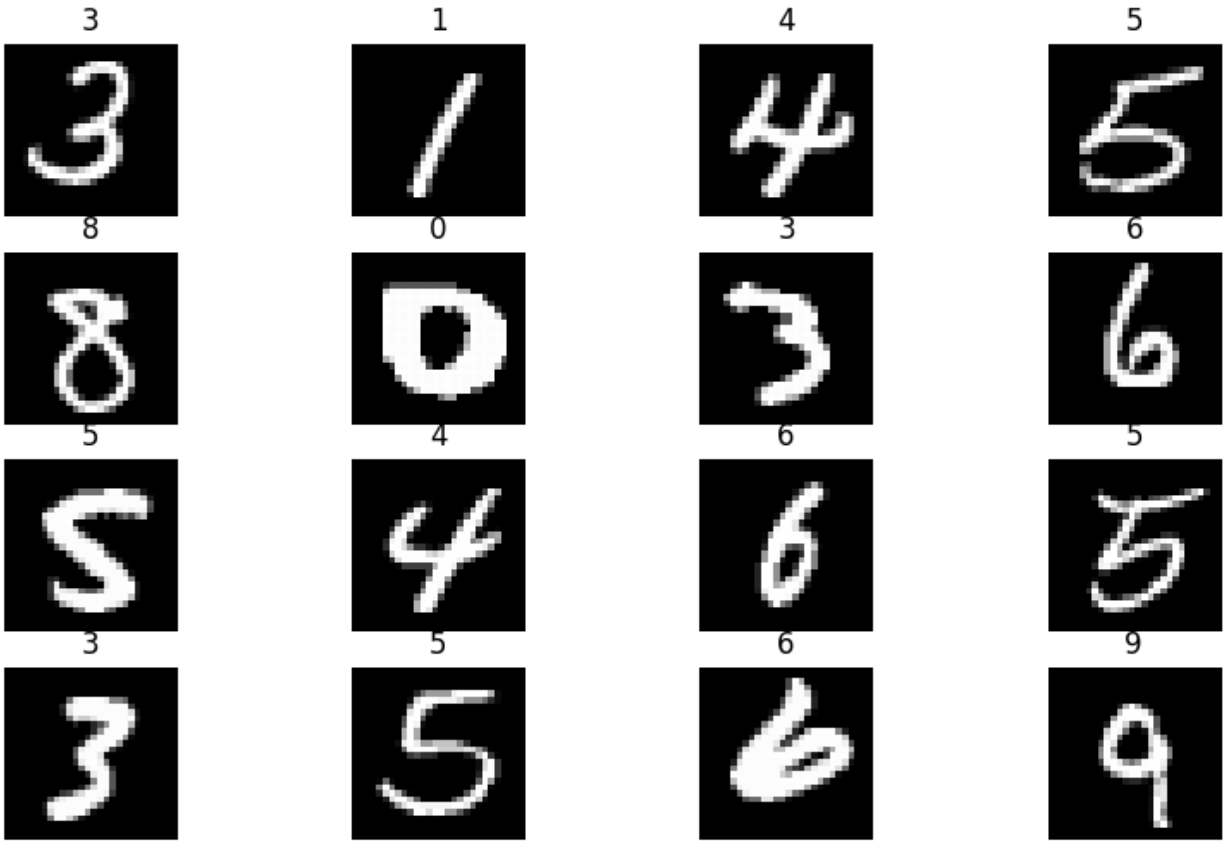
The images in MNIST are gray images, and therefore, they have only one channel. To check the shape of the images we can use `size()` in PyTorch which corresponds to `shape` in NumPy.

[ ]:
```
print(training_data.data.size())
print(test_data.data.size())
```

```
torch.Size([60000, 28, 28])
torch.Size([10000, 28, 28])
```

Let's visualize several randomly selected images from the training dataset, and show their class labels.

[ ]:
```
figure = plt.figure(figsize=(10, 6))
cols, rows = 4, 4
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(training_data), size=(1,)).item()
    img, label = training_data[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(label)
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```



DataLoader

In PyTorch, `DataLoader` is an iterable object used for passing a batch of input data at each iteration during the model training. `DataLoader` takes as arguments the dataset that we are going to use, `shuffle` indicates if the data should be shuffled, and `batch_size` is self-explanatory. While we always need to shuffle the data in the training dataset to avoid correlated data (e.g., the training dataset can have first all 0 images, then 1 images, etc.), we don't need to shuffle the test or validation datasets because they are not used for training.

[ ]:
```
train_dataloader = DataLoader(training_data, shuffle=True, batch_size=128)
test_dataloader = DataLoader(test_data, shuffle=False, batch_size=128)
```

The output of DataLoader is a batch of input images and the corresponding target labels. To inspect a batch of data we need to use the `iter` method to obtain an iterator, and afterward we can use the `next()` function to iterate over the batches. Note that `DataLoader` converted the images into a format (1, 28, 28) where 1 is the number of channels of the images.

Note also that in PyTorch the default convention for representing the dimensions of images in tensors is the "channels first" format as in (1, 28, 28), as opposed to Keras-TensorFlow where the default convention is the "channels last" format as in (28, 28, 1).

[ ]:
```
# check is train_dataloader is an iterator
iter(train_dataloader) is train_dataloader
```

```
False
```

[ ]:
```
# Inspect a batch of images and labels
batch_images, batch_labels = next(iter(train_dataloader))
print("Batch images shape:", batch_images.size())
print("Batch labels shape:", batch_labels.size())
```

```
Batch images shape: torch.Size([128, 1, 28, 28])
Batch labels shape: torch.Size([128])
```

7.18.3 18.3 Training Neural Networks: Revisited

Before explaining how to create and train NNs with PyTorch, let's briefly review the section of training NN from Lecture 15, as it will be helpful to understand the PyTorch code.

Training NNs is performed by iterative updates of the model parameters with the goal to minimize the difference between the model predictions and target labels.

Each iteration in the training phase includes the following 4 steps:

1. Forward pass (forward propagation)
2. Loss calculation
3. Error backpropagation (backward pass)
4. Model parameters update

Forward propagation or **forward pass**, involves passing the input data through all hidden layers of the neural network toward the output layer to obtain the network predictions. If the input data is an image, the image is transformed through the layers of the network, and for classification problems, the output is a vector of predicted class probabilities.

Loss calculation is the second step, in which the loss of the network is calculated as a difference between the network predictions and the target labels. For classification tasks, standard loss function is crossentropy loss, and for regression tasks loss functions include mean-squared deviation and mean absolution deviation.

Error backpropagation, also called **backward pass** or **backward propagation** involves propagating the predicted outputs back through the network, from the last layer backward toward the first layer. During the backward step, the gradients of the loss with respect to the model parameters $\nabla\mathcal{L}()$ are calculated. The gradients quantify the impact of changing the parameters in the network to the predicted outputs. Automatic calculation of the gradients (automatic differentiation) is available in all current deep learning libraries, which significantly simplifies the implementation of deep learning algorithms.

Model parameters update is the last step in which new values for the model parameters are calculated and updated, typically using the **Gradient Descent** algorithm.

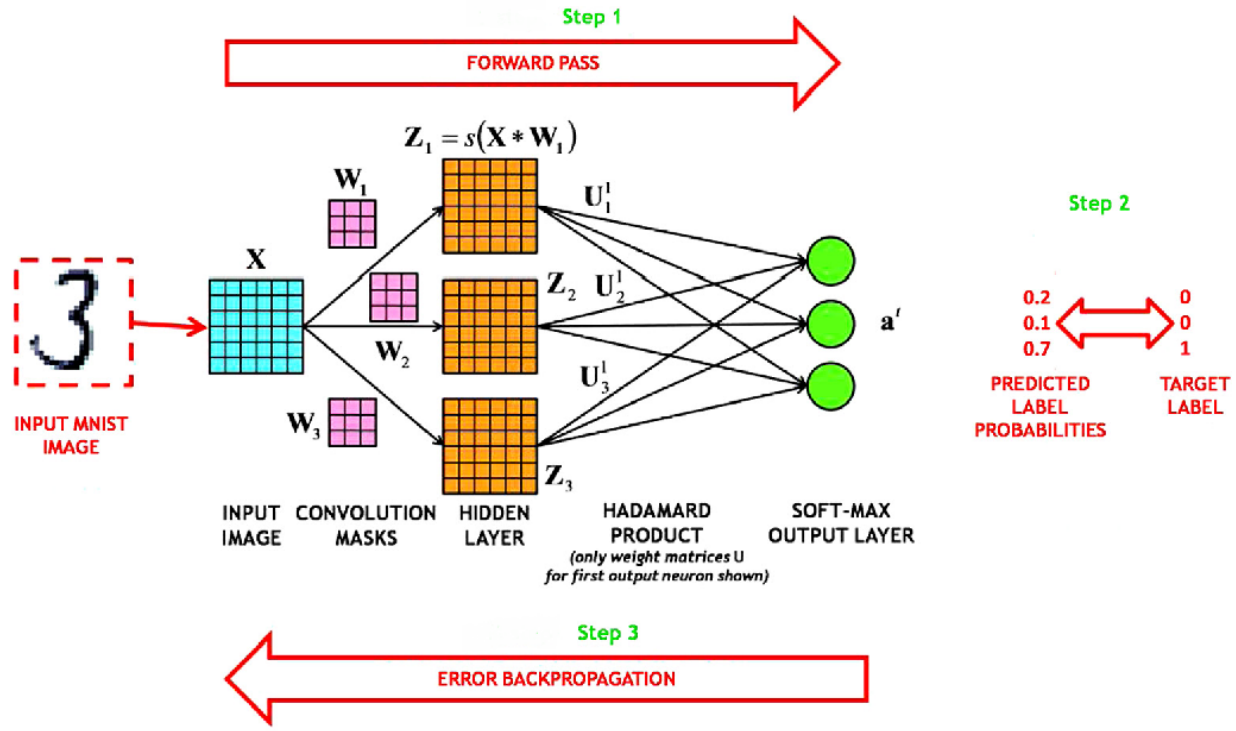
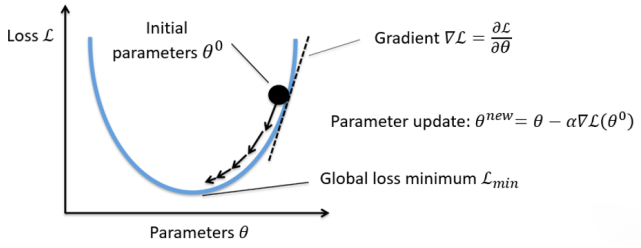


Figure: Steps in training neural networks.

And to briefly review the **Gradient Descent** algorithm, it uses the gradient of the loss function to estimate the slope of the loss function. By updating the parameters in the opposite direction of the gradient of the loss $\nabla\mathcal{L}$, the algorithm finds parameters for which the loss $\mathcal{L}$ has a minimal value.



Gradient descent algorithm:

1. Randomly initialize the parameters θ^0
2. Compute the gradient of the loss function: $\nabla\mathcal{L}(\theta^0)$
3. Update the parameters: $\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$
   - α is learning rate
4. Go to step 2 and repeat

Figure: Gradient descent algorithm.

Almost all modern neural networks are trained by applying a modified version of the Gradient Descent algorithm. Examples of such advanced Gradient Descent algorithms include Adam, SGD (Stochastic Gradient Descent), RMSprop, Adagrad, Nadam, and others.

To train a neural network, the steps of forward pass, loss calculation, backward pass, and parameter update are performed iteratively for each batch of the input data. Each iteration through the input data constitutes one epoch. This is

shown in the following simple pseudocode.

```
for epoch in number_of_epochs:
    for batch in number_of_batches:
        forward pass
        calculate loss
        backward pass (calculate the gradients)
        update parameters
```

For predicting on test data (*inference*) only a forward pass through the model is needed, and it does not require to calculate the loss, perform the backward pass, or update the model parameters.

7.18.4 18.4 Creating, Training, and Evaluating the Model

Model Definition

We will define a Convolution Neural Network using the `nn.Module` in PyTorch, which is a superclass for creating neural networks in PyTorch. I.e., the CNN model will represent a subclass of the superclass `nn.Module`. Inheriting from the `nn.Module` allows to use various PyTorch functionalities for our model. Let's use the name `CNN` for the subclass that we will use to instantiate our model.

In the `__init__()` constructor method of the CNN class, we will list the layers of the network as attributes of the class. For this task, let's use 2 convolutional layers that we will name `conv1` and `conv2`, a max-pooling layer `maxpool`, ReLU layer `relu`, and a dense layer named `dense`. The `nn.Module` also offers several other attributes and methods which we will inherit via `super().__init__()` (or we can also use `nn.Module.__init__(self)` to achieve the same result, as you may recall from the lecture on OOP).

Conv2d layers in PyTorch have similar arguments to the Keras library, as follows:

- *in_channels* (int), number of input channels to the layer, which for the first layer in the network is the number of channels in input images, and for all other layers is the number of output channels from the preceding layer (recall that the `in_channels` argument is not provided in the Keras layers and it is determined automatically, however in PyTorch we need to specify it).
- *out_channels* (int), number of channels that are produced by the layer (i.e., the number of used convolutional filters).
- *kernel_size* (int or tuple), size of the convolving kernel (it is typically 3, and sometimes can also be 5, 7, etc.).
- *padding* (int or tuple, optional), padding can be added to both sides of the input images; default value is 0, i.e., no padding.

Regarding the image padding, note that applying a convolutional filter to an image produces a convolved image with a reduced size. To obtain a resulting feature map of the same size as the original image, padding is applied. For instance, in the figure below, a filter with a kernel size 3x3 is applied to a padded image of size 7x7, and the output is a feature with size 5x5. In this case, the original image of size 5x5 is padded with zeros on all four sides, and the size was changed to 7x7. In PyTorch `padding=1` means that one row or column is added on all sides of the image. This is equivalent to `padding='same'` in Keras.

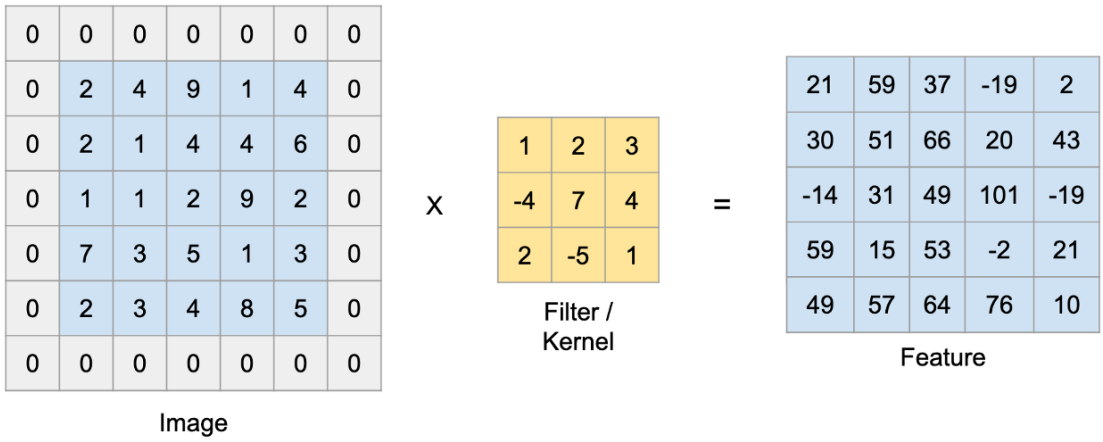


Figure: Image padding.

Max pooling layers are defined the same as in Keras, and have as argument the kernel size for the pooling operation. Most networks use kernel size of 2 for the pooling operation.

The outputs of convolutional layers are passed through a **ReLU activation layer** which is short for **Re**ctified **L**inear **U**nit activation function. As we explained, activation functions introduce non-linearity to the layers in the model, which allows to learn complex relations between the inputs and outputs. Most modern neural networks apply ReLU activation function, or some variants of it ReLU activation function is shown in the next figure, and it outputs 0 if the input is negative, or outputs the input if it is positive.

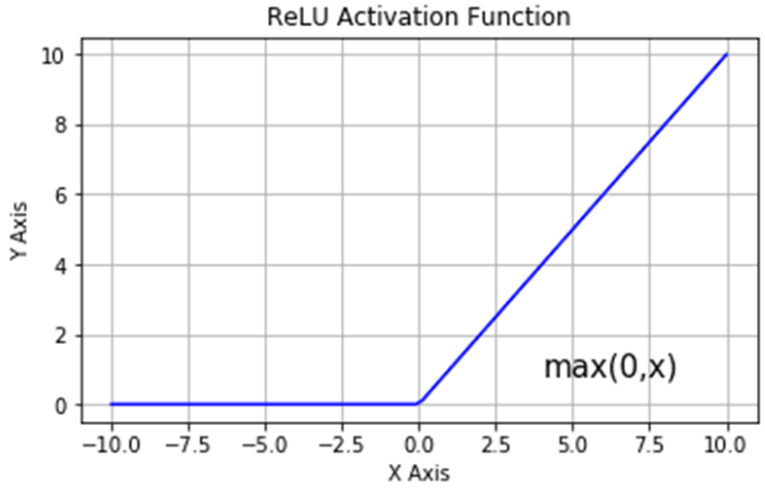


Figure: ReLU activation function.

Fully-connected (dense) layers in PyTorch are defined with the `nn.Linear` layer, and they have as arguments:

- *in_features* (int), number of input features to the layer (this argument is not provided in the Keras Dense layer, it is determined automatically).
- *out_features* (int), number of output features of the layer.

After we define the layers in the model, we will add the method `forward` to define the forward pass for the model. For this model, we will use 2 blocks of convolutional, ReLU, and max-pooling layers, and a final dense layer to make the predictions. Also recall that in the previous lecture we used the Flatten layer to convert the outputs from the convolutional layers into one-dimensional vectors. Here, we use `torch.flatten()` to flatten the tensors.

[ ]:
```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()

        # convolutional layers
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)

        # maxpooling layer
        self.maxpool = nn.MaxPool2d(kernel_size=2)

        # ReLU activation layer
        self.relu = nn.ReLU()

        # fully connected layer, output 10 classes
        self.dense = nn.Linear(in_features=32 * 7 * 7, out_features=10)

    def forward(self, x):
        # sequence of layers
        # first block
        x = self.conv1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        # second block
        x = self.conv2(x)
        x = self.relu(x)
        x = self.maxpool(x)

        # flatten the output of the second block to 1D vector
        x = torch.flatten(x, 1)

        # output layer
        output = self.dense(x)

        return output
```

Next, we will create an instance of the model, here named `cnn_model`.

[ ]:
```
cnn_model = CNN()
```

In PyTorch we need to specify the device on which the model will be trained, i.e., whether we will use CPU, GPU, or TPU. We can do that with `torch.device`, as in the next cell.

I am using Google Colab with GPU available, therefore in the output of the cell the device type is `'cuda'`.

CUDA (Compute Unified Device Architecture) is a library that allows using GPU computing for machine learning tasks, which parallelizes the computations, and speeds up the model training.

[ ]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device
```

```
device(type='cuda')
```

Using `to(device)` will transfer the ConvNet model to the device, which in our case is GPU. And later, we will also

move the data for training the model to the GPU.

[ ]:
```
cnn_model.to(device)
```

```
CNN(
  (conv1): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu): ReLU()
  (dense): Linear(in_features=1568, out_features=10, bias=True)
)
```

Model Training

Before training the model, we need to define a loss function and an optimizer. As we mentioned in the previous lecture, crossentropy loss is commonly used with classification problems, and Adam is a standardly used optimization algorithm.

[ ]:
```
# define loss function (cross-entropy)
criterion = nn.CrossEntropyLoss()
# define an optimizer (Adam)
optimizer = optim.Adam(cnn_model.parameters(), lr = 0.01)
```

The above loss is equivalent to the `sparse_categorical_crossentropy` in Keras-TensorFlow. Therefore, if the target variable is in one-hot encoding format, we will need to first convert into 1D tensor containing the integer class labels. Similarly, if we would like to perform binary classification in PyTorch we can specify a binary crossentropy loss with `nn.BCELoss`.

The following cell trains the model, and it is similar to the `fit` function in scikit-learn or Keras. It contains a for-loop over the epochs, and afterwards the variables `running_loss`, `total`, and `sum_correct` are initialized and will be used to store the values of the training loss and accuracy at each epoch.

Next, a batch of images is loaded, and moved to the device (GPU in this case). And, the above four steps are performed: forward pass, loss calculation, backward pass, and parameters update.

The remaining lines calculate the loss and accuracy and display their values after each epoch.

The `item()` method is used to convert the tensors to NumPy floats and bring them to the CPU. PyTorch uses GPU for performing calculations, and CPU for displaying and plotting the variables.

Please read the comments in the next cell to better understand the code.

[ ]:
```
# total number of training epochs
epoch_num = 10

# loop over the number of epochs
for epoch in range(epoch_num):

    # the loss values for each epoch
    running_loss = 0.0
    # total images in each epoch
    total = 0
    # correctly predicted images in each epoch
    sum_correct = 0

    ### training
```

(continues on next page)

(continued from previous page)

```
    # loop over the batches in the training dataset
    for i, data in enumerate(train_dataloader):
        # get a batch of images and labels
        images, labels = data
        # send the images and labels to the GPU
        images, labels = images.to(device), labels.to(device)

        # set the gradient to zero, to clear the values from the last iteration
        optimizer.zero_grad()

        # forward pass: propagate the inputs through the network to obtain output␣
↪predictions
        outputs = cnn_model(images)
        # calculate the loss (crossentropy)
        loss = criterion(outputs, labels)
        # backward pass: propagate backward and calculate the gradients
        loss.backward()
        # update the model parameters (using the optimizer Adam)
        optimizer.step()

        # calculate the loss and accuracy for the current batch
        # 'loss' of the current batch is added to the 'running_loss'
        # '.item()' returns the value of the PyTorch tensor as Python float32
        # and moves its value to the CPU
        running_loss += loss.item()
        # 'predicted' is the vector of class labels for images in the batch
        # this is similar to np.argmax
        _, predicted = torch.max(outputs.data, dim=1)
        # correct is the number of correctly predicted images in the batch
        sum_correct += (predicted==labels).sum().item()
        # 'total' is the number of images (or labels) in the batch
        total += labels.size(0)

    # calculate the accuracy for each epoch
    accuracy = sum_correct/total

    # print the epoch number, training loss, and training accuracy
    print(f'Epoch: {epoch + 1}/{epoch_num}\t Training loss: {running_loss:.3f}\t',
              f'\t Training accuracy: {100*accuracy:2.3f}')
```

```
Epoch: 1/10      Training loss: 107.009          Training accuracy: 93.078
Epoch: 2/10      Training loss: 36.236           Training accuracy: 97.613
Epoch: 3/10      Training loss: 30.805           Training accuracy: 97.912
Epoch: 4/10      Training loss: 26.868           Training accuracy: 98.237
Epoch: 5/10      Training loss: 25.493           Training accuracy: 98.338
Epoch: 6/10      Training loss: 24.083           Training accuracy: 98.435
Epoch: 7/10      Training loss: 24.274           Training accuracy: 98.323
Epoch: 8/10      Training loss: 22.736           Training accuracy: 98.438
Epoch: 9/10      Training loss: 21.969           Training accuracy: 98.493
Epoch: 10/10     Training loss: 21.570           Training accuracy: 98.602
```

Model Evaluation

The following cell evaluates the accuracy on the test dataset. For this step, we will use `with torch.no_grad()` context manager, to indicate that there is no need to calculate the gradients, since the model parameters are not updated during model evaluation. As we know, only the forward pass is required for evaluation on test images.

[ ]:
```
### testing

# these variables are similar to the variables for the training phase
test_running_loss = 0.0
test_total = 0
test_correct = 0

# torch.no_grad() specify not to update the model during testing
with torch.no_grad():
  # loop over the batches in the test dataset
  for i, data in enumerate(test_dataloader):
      # get a batch of images and labels
      images, labels = data
      # send the images and labels to the GPU
      images, labels = images.to(device), labels.to(device)

      # forward pass
      outputs = cnn_model(images)
      # calculate the loss
      loss = criterion(outputs, labels)
      # there is no backward pass in the testing step

      # these variables are the same as in the training setp
      test_running_loss += loss.item()
      _, predicted = torch.max(outputs.data, 1)
      test_total += labels.size(0)
      test_correct += (predicted == labels).sum().item()

print(f'Accuracy of the model on the test images: {100*test_correct/test_total:2.3f}')
```

```
Accuracy of the model on the test images: 98.230
```

7.18.5 18.5 Using a Custom Dataset and a Pretrained Model

Loading the Dataset

In this section, we will study one more classification task with CNNs in PyTorch. We will use the image dataset `Imagenette`, which is just a small subset of images from the large dataset ImageNet. There are a few versions of the `Imagenette` dataset, and we will use a version that has 9,469 images, categorized into 10 classes. The classes are: tench, English springer, cassette player, chain saw, church, French horn, garbage truck, gas pump, golf ball, and parachute.

We will explore two different cases for the organization of a custom dataset to be loaded:

- Case 1, the dataset consists of separate directories for each class, where each subdirectory contains the data samples for one class
- Case 2, the dataset has one directory with all data samples and a spreadsheet (or text file) that contains the labels for all samples

Case 1: One Directory for Each Class

As we stated, in this case, the dataset is organized into multiple directories, and each directory contains the data samples for one class.

Let's first mount the Google Drive since the dataset is saved on the drive, and uncompress the file.

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
# Uncompress the dataset
!unzip -uq "drive/MyDrive/Data_Science_Course/Fall_2023/Lecture_18-NNs_with_PyTorch/data/
↪imagenette_folders.zip" -d "sample_data/"
```

To see the dataset, click on the icon that looks like a folder located in the left-side panel in Google Colab, and under `sample_data` you will see the `imagenette_folders` directory. If we click on the arrow to expand it, we can see that it has two main subdirectories `train` and `val`, each of which has 10 subdirectories that contain the images for each class in the dataset.

in PyTorch, loading datasets that have such organization is very simple. The function `ImageFolder` is designed for such datasets, and it automatically labels and organizes the data. In the cell below, we just provide the path to the dataset, and `transform` for resizing the images to 128x128 pixel size and converting the data to PyTorch tensors.

After that, we use the `DataLoader` to create train and validation dataloader objects for iterating over the datasets.

[ ]:
```
from torchvision.datasets import ImageFolder
from torchvision import transforms

transform = transforms.Compose([transforms.Resize((128, 128)), transforms.ToTensor()])

train_dataset = ImageFolder(root='sample_data/imagenette_folders/train/',␣
↪transform=transform)
val_dataset = ImageFolder(root='sample_data/imagenette_folders/val', transform=transform)

train_dataloader = DataLoader(train_dataset, shuffle=True, batch_size=64)
val_dataloader = DataLoader(val_dataset, shuffle=False, batch_size=64)
```

Case 2: All Data Points in a Single Directory

In this case, all images are saved into one single directory. Let's uncompress the file. Again, if we click on the folder icon in the left-side panel in Google Colab, we can notice that the dataset has one directory `images` that contains all images, and a file `labels.csv` that has the labels for all images.

[ ]:
```
# Uncompress the dataset
!unzip -uq "drive/MyDrive/Data_Science_Course/Fall_2023/Lecture_18-NNs_with_PyTorch/data/
↪imagenette_all.zip" -d "sample_data/"
```

Custom datasets in PyTorch are created as a subclass of the `Dataset` class, which is imported from `torch.utils.data`. The custom datasets should inherit from `Dataset` and they are required to override the following two methods:

- `__len__()`, to return the number of instances in the dataset.
- `__getitem__()` to support indexing and return the instances with the specified index `idx` in the dataset.

In the cell below, we used `MyDataset` as the name of the subclass. In the `__init__()` method we defined the root directory where the dataset is located, the subdirectory `image_folder` where the images are saved, we read the `labels.csv` file as a Pandas DataFrame, and we defined `transform` that will apply transformation to the images.

The `__len__()` method returns the length of the dataset, which is the same as the number of rows in `labels.csv`.

In the `__get_item__()` method, `idx` is the index of the data point to load, `img_name` is the file path to the image with index idx (where the image files are named 'img_0001.jpg', 'img_0002.jpg', etc.), `image` is the image file and it is loaded with the `Image.open()` method in PIL (Python Imaging Library), and afterward transformations are applied to the image. The `label` for each image is extracted from the Pandas DataFrame `labels_file`. Finally, the `__getitem__()` method returns a tuple containing the transformed image and its corresponding label.

[ ]:
```
class MyDataset(Dataset):
    def __init__(self, root_dir, transform):
        self.root_dir = root_dir
        self.image_folder = os.path.join(root_dir, 'images')
        self.labels_file = pd.read_csv(os.path.join(root_dir, 'labels.csv'), header=None)
        self.transform = transform

    def __len__(self):
        return len(self.labels_file)

    def __getitem__(self, idx):
        img_name = os.path.join(self.image_folder, f'img_{idx + 1:04}.jpg')
        image = Image.open(img_name)
        image = self.transform(image)
        label = self.labels_file.iloc[idx, 0]

        return image, label
```

Now, let's use the class `MyDataset` to instantiate the dataset. There are 9,469 images in total in the dataset.

[ ]:
```
transform = transforms.Compose([transforms.Resize((128, 128)), transforms.ToTensor()])

dataset = MyDataset(root_dir='sample_data/imagenette_all', transform=transform)
```

[ ]:
```
print(len(dataset))
```

```
9469
```

To create training, testing, and validation subsets, we will first use scikit-learn's `train_test_split` to split the indices into three groups.

Afterward, we will use the `Subset` class in PyTorch to partition the dataset into training, testing, and validation datasets based on the indices.

And, as in the above examples, we will use `DataLoader` to enable iterating over batches of images and labels.

[ ]:
```
from sklearn.model_selection import train_test_split

# Split the dataset into train and test subsets
train_indices_1, test_indices = train_test_split(range(len(dataset)), test_size=0.2,␣
↪random_state=123)

# Split the dataset into train and validation subsets
train_indices, val_indices = train_test_split(train_indices_1, test_size=0.2, random_
↪state=123)
```

(continues on next page)

(continued from previous page)

[ ]:
```
# Create training, testing, and validation subsets
train_dataset = Subset(dataset, train_indices)
test_dataset = Subset(dataset, test_indices)
val_dataset = Subset(dataset, val_indices)
```

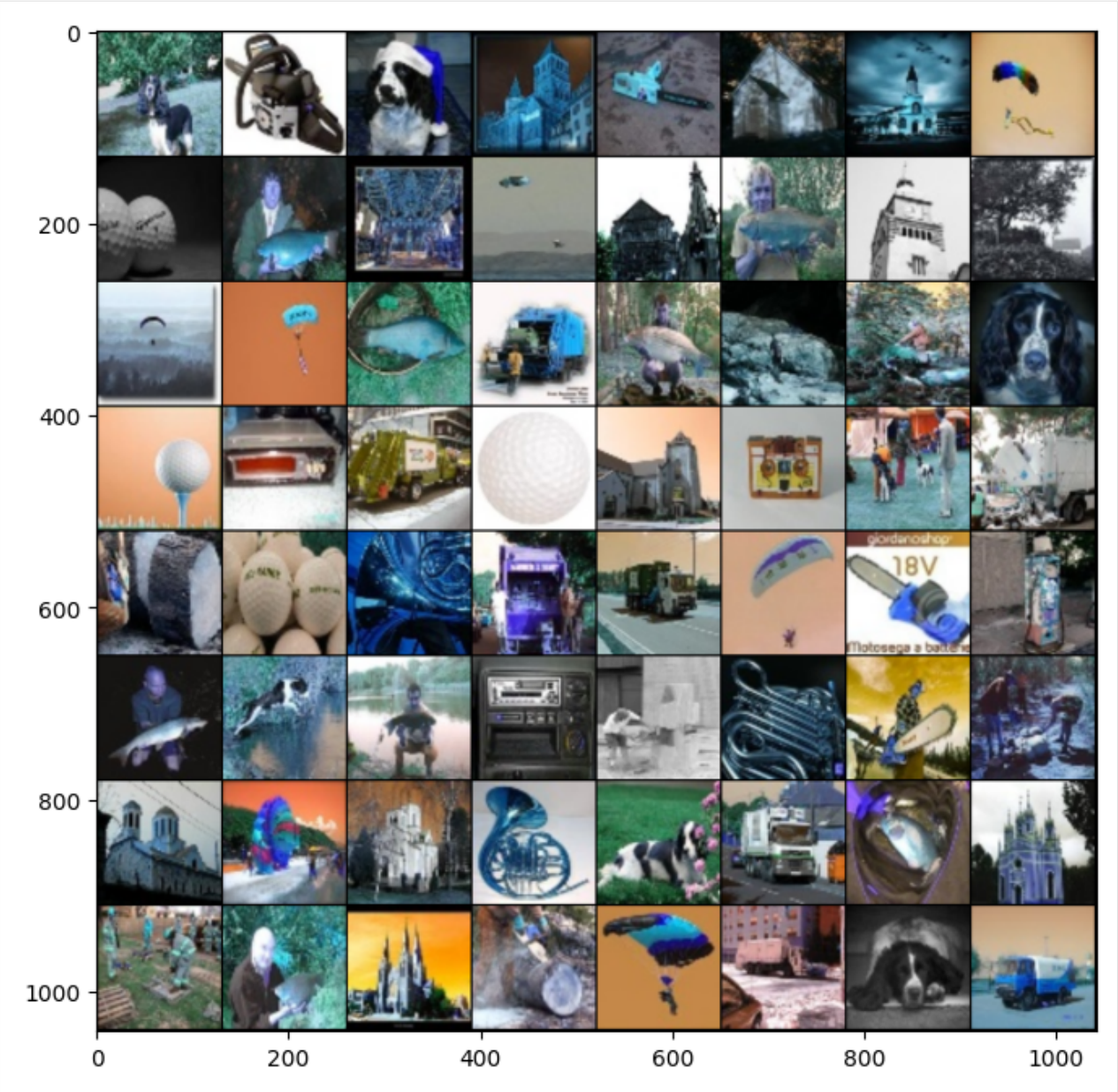
[ ]:
```
# Create training, testing, and validation dataloaders for iterating over the datasets
train_dataloader = DataLoader(train_dataset, shuffle=True, batch_size=64)
test_dataloader = DataLoader(test_dataset, shuffle=False, batch_size=64)
val_dataloader = DataLoader(val_dataset, shuffle=False, batch_size=64)
```

[ ]:
```
print(len(train_dataset))
print(len(test_dataset))
print(len(val_dataset))
```

```
6060
1894
1515
```

PyTorch has a function `make_grid()` which allows to plot a grid of images.

[ ]:
```
# show several images
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# get a batch of random training images and labels
images, labels = next(iter(train_dataloader))

# show images
plt.figure(figsize=(8,8))
imshow(torchvision.utils.make_grid(images))
```



Model Loading

PyTorch has pretrained models that are easy to be loaded and used. Let's import a VGG model pretrained on the ImageNet dataset.

[ ]:
```
from torchvision.models import vgg16

#VGG16
vgg_model = vgg16(pretrained=True)
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning:␣
↪The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future,␣
↪please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning:␣
↪Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13␣
↪and may be removed in the future. The current behavior is equivalent to passing␣
↪`weights=VGG16_Weights.IMAGENET1K_V1`. You can also use `weights=VGG16_Weights.
↪DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.cache/
↪torch/hub/checkpoints/vgg16-397923af.pth
100%|| 528M/528M [00:07<00:00, 71.6MB/s]
```

To use the VGG model with our dataset, we will change the last layer in the model with a dense layer that will output 10 features, since there are 10 classes in the dataset. Notice in the following cell which shows the architecture of the VGG model that the last layer is layer '6' in the classifier (where the classifier refers to the Linear (dense), ReLU, and Droput layers that follow the convolutional and maxpooling layers). In the original VGG model that was trained on ImageNet, the last layer has 1,000 output neurons, since there are 1,000 classes in ImageNet.

[ ]:
```
# replace the last layer in VGG with a dense layer to predict 10 classes
vgg_model.classifier._modules['6'] = nn.Linear(4096, 10)
```

[ ]:
```
vgg_model.to(device)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
```

(continues on next page)

(continued from previous page)

```
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

Modular Code: Training and Validation for One Epoch

For this example, let's write functions for training and validating the model which we will call `train` and `validate`. These functions will group all required code for training and validation, and we can re-use such modular code in other scripts.

The functions are almost identical to the code that we used above. They return the accuracy and loss for the epoch. Note that we used the lines `model.train()` and `model.eval()` in these functions, which act as a switch for some specific layers that behave differently during training and evaluation. For example, Dropout and Batch Normalization layers are turned on during training and turned off during evaluation, which is controlled by these two lines.

[ ]:
```python
# train the model for one epoch on the given set
def train(model, train_loader, criterion, optimizer, epoch):
    running_loss, total, sum_correct = 0.0, 0, 0

    # indicate this is a training step
    model.train()

    for i, data in enumerate(train_loader):
        images, labels = data
        labels = labels.type(torch.LongTensor)
        images, labels = images.to(device), labels.to(device)

        # forward + loss + backward + update
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # calculate loss and accuracy
        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
```

(continues on next page)

(continued from previous page)

```
        sum_correct += (predicted == labels).sum().item()
        total += labels.size(0)

    # return the accuracy and loss
    return sum_correct/total, running_loss


# evaluate the model on the given set
def validate(model, val_loader, criterion):
    running_loss, total, sum_correct = 0.0, 0, 0

    # indicate this is an evaluation step
    model.eval()

    with torch.no_grad():
        for i, data in enumerate(val_loader):
            images, labels = data
            labels = labels.type(torch.LongTensor)
            images, labels = images.to(device), labels.to(device)

            # Compute the output: forward pass only
            outputs = model(images)
            loss = criterion(outputs, labels)

            # calculate loss and accuracy
            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            sum_correct += (predicted == labels).sum().item()

    # return the accuracy and loss
    return sum_correct/total, running_loss
```

Define Loss and Optimizer

[ ]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(vgg_model.parameters(), lr=1e-4)
```

Model Training

The next cell contains code for training the model. Now we can call the `train` and `validate` functions to perform the training and validation steps for each epoch. The outputs are the average accuracy and loss, calculated at the end of each epoch. We will append those values to lists, which we will use later to plot the learning curves.

[ ]:
```
# total number of training epochs
epoch_num = 10

# initialize variables to save the training and validation loss and accuracy
training_loss_plot = []
```

(continues on next page)

(continued from previous page)

```
training_accuracy_plot = []
val_loss_plot = []
val_accuracy_plot = []

# loop over the number of epochs
for epoch in range(epoch_num):

    # train for one epoch: return accuracy and loss
    tr_accuracy, tr_loss = train(vgg_model, train_dataloader, criterion, optimizer,␣
↪epoch)

    # evaluate after each epoch: return accuracy and loss
    val_accuracy, val_loss = validate(vgg_model, val_dataloader, criterion)

    # append the accuracies and losses after each epoch
    training_accuracy_plot.append(tr_accuracy)
    training_loss_plot.append(tr_loss)
    val_accuracy_plot.append(val_accuracy)
    val_loss_plot.append(val_loss)

    # Display after each epoch
    print(f'Epoch: {epoch + 1}/{epoch_num}\t Training loss: {tr_loss:.3f}\t',
              f'Training accuracy: {100*tr_accuracy:2.3f}\t Validation accuracy:
↪{100*val_accuracy:2.3f}')
```

```
Epoch: 1/10      Training loss: 69.309   Training accuracy: 75.644       Validation␣
↪accuracy: 88.977
Epoch: 2/10      Training loss: 22.932   Training accuracy: 92.277       Validation␣
↪accuracy: 88.977
Epoch: 3/10      Training loss: 11.393   Training accuracy: 95.974       Validation␣
↪accuracy: 89.703
Epoch: 4/10      Training loss: 8.982    Training accuracy: 97.013       Validation␣
↪accuracy: 86.799
Epoch: 5/10      Training loss: 5.881    Training accuracy: 98.036       Validation␣
↪accuracy: 91.089
Epoch: 6/10      Training loss: 5.154    Training accuracy: 98.218       Validation␣
↪accuracy: 89.703
Epoch: 7/10      Training loss: 6.606    Training accuracy: 97.987       Validation␣
↪accuracy: 90.363
Epoch: 8/10      Training loss: 5.586    Training accuracy: 98.218       Validation␣
↪accuracy: 89.835
Epoch: 9/10      Training loss: 2.577    Training accuracy: 99.092       Validation␣
↪accuracy: 89.241
Epoch: 10/10     Training loss: 1.870    Training accuracy: 99.356       Validation␣
↪accuracy: 89.109
```

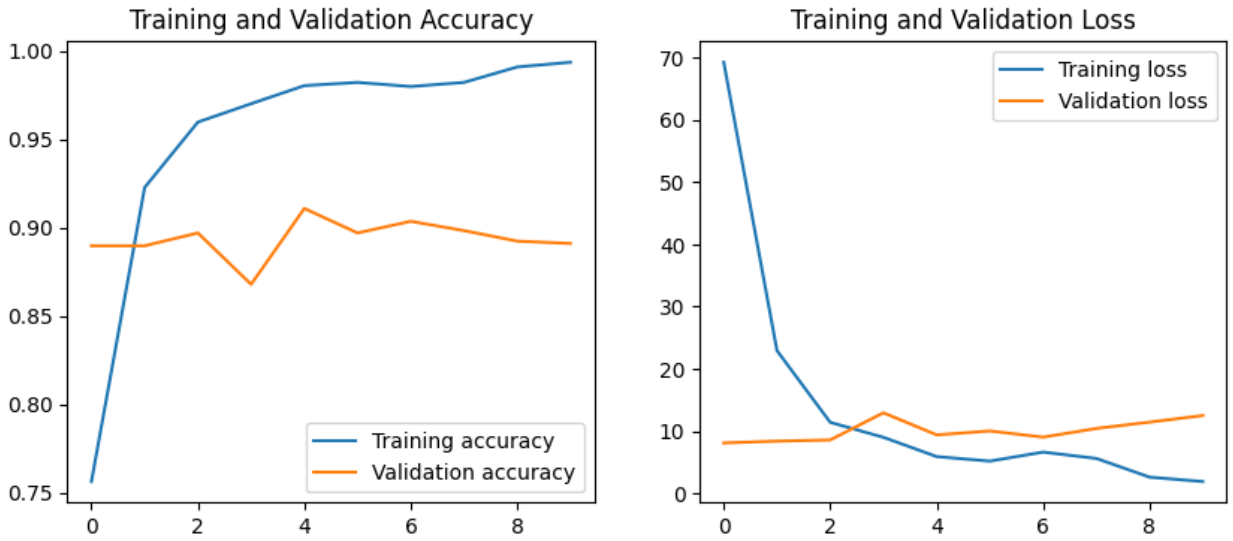
The training and validation accuracies and loss are shown in the next figure. Note that Mathplotlib doesn't support plotting tensors that are on the GPU, and therefore, we needed to bring the values of the accuracies and losses to the CPU for plotting (e.g., by using `item()` as in the above code).

[ ]:
```
# plot the accuracy and loss for the training and validation datasets
plt.figure(figsize=(10, 4))
plt.subplot(1,2,1)
```

(continues on next page)

(continued from previous page)

```
plt.plot(training_accuracy_plot)
plt.plot(val_accuracy_plot)
plt.legend(['Training accuracy', 'Validation accuracy'])
plt.title('Training and Validation Accuracy')
plt.subplot(1,2,2)
plt.plot(training_loss_plot)
plt.plot(val_loss_plot)
plt.legend(['Training loss', 'Validation loss'])
plt.title('Training and Validation Loss')
plt.show()
```



Model Evaluation

We can use the same function `validate` to evaluate the model performance on the test dataset.

```
[ ]: # calculate the accuracy and loss on the test dataset
test_accuracy, test_loss = validate(vgg_model, test_dataloader, criterion)
print(f'Test dataset accuracy: {100*test_accuracy:2.3f}')
```

```
Test dataset accuracy: 89.335
```

7.18.6 18.6 Model Saving and Loading in PyTorch

Save and Load State Dictionary

To save PyTorch models we use `torch.save()` as shown below. The parameters of the model are stored in a state dictionary with `state_dict()`, and as expected, the path to the directory to save the model needs to be provided. A `state_dict()` is a dictionary that has as keys the layers in the network, and the values in the dictionary are the corresponding parameters in each layer.

A PyTorch convention is to use the extension `.pth` or `.pt` with the file path.

[ ]:
```
torch.save(vgg_model.state_dict(), 'vgg_model_weights.pth')
```

Let's print the keys in the `state_dict` for the vgg model. The model has 30 layers in the `features` part, and 6 layers in the `classifier` part (as shown in one of the above cells). E.g., `features.0.weight` is the key for the weights in layer 0 and `features.0.bias` is the key for the biases in layer 0 of the model, where biases is a vector of trainable constant values that are added to the weights in each layer.

[ ]:
```
print("Keys in the state_dict keys: \n", vgg_model.state_dict().keys())
```

```
Keys in the state_dict keys:
 odict_keys(['features.0.weight', 'features.0.bias', 'features.2.weight', 'features.2.
↪bias', 'features.5.weight', 'features.5.bias', 'features.7.weight', 'features.7.bias',
↪'features.10.weight', 'features.10.bias', 'features.12.weight', 'features.12.bias',
↪'features.14.weight', 'features.14.bias', 'features.17.weight', 'features.17.bias',
↪'features.19.weight', 'features.19.bias', 'features.21.weight', 'features.21.bias',
↪'features.24.weight', 'features.24.bias', 'features.26.weight', 'features.26.bias',
↪'features.28.weight', 'features.28.bias', 'classifier.0.weight', 'classifier.0.bias',
↪'classifier.3.weight', 'classifier.3.bias', 'classifier.6.weight', 'classifier.6.bias
↪'])
```

To load a saved model, we need to have an instance of the model, and therefore let's create an instance `model_1`. Then, we use the `load_state_dict()` method with `torch_load()`. Internally PyTorch uses pickle to serialize and deserialize the `state_dict()` when saving and loading.

[ ]:
```
model_1 = vgg_model

model_1.load_state_dict(torch.load('vgg_model_weights.pth'))
```

```
<All keys matched successfully>
```

[ ]:
```
# calculate the accuracy and loss on the test dataset
test_accuracy, test_loss = validate(model_1, test_dataloader, criterion)
print(f'Test dataset accuracy: {100*test_accuracy:2.3f}')
```

```
Test dataset accuracy: 89.335
```

Save and Load the Entire Model

Instead of saving only the model parameters with the `state_dict()`, it is also possible to save the entire model in PyTorch, including the class definition, the architecture with the layers, and other related information, as in the next cell.

[ ]:
```
torch.save(vgg_model, 'vgg_model_2.pth')
```

However, saving the state dictionary is the preferred way, since it results in a reduced file size, and it allows to share models accross different versions of PyTorch, whereas saved objects with the entire model may not be compatible between PyTorch versions. In addition, loading a saved state dictionary provides more flexibility and it allows to replace the model achitecture and fine-tune the model with a different architecture.

Save and Load a Checkpoint

If we would like to save the model checkpoint after an epoch, and later resume the training, we will need to also save information about the loss and gradients at that epoch, along with the state dictionary. In the next cell, the checkpoint saves the last epoch, the state dictionary `model.state_dict()`, and the optimizer information `optimizer.state_dict()`.

[ ]:
```
checkpoint = {'epoch': epoch,
              'state_dict': vgg_model.state_dict(),
              'optimizer_state' : optimizer.state_dict()}

torch.save(checkpoint, 'vgg_checkpoint.pth')
```

When loading the checkpoint, we will load the `'state_dict'`, `'optimizer_state'`, and `'epoch'`.

[ ]:
```
model_2 = vgg_model

checkpoint = torch.load('vgg_checkpoint.pth')
model_2.load_state_dict(checkpoint['state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state'])
epoch = checkpoint['epoch']
```

[ ]:
```
# calculate the accuracy and loss on the test dataset
test_accuracy, test_loss = validate(model_2, test_dataloader, criterion)
print(f'Test dataset accuracy: {100*test_accuracy:2.3f}')
```

```
Test dataset accuracy: 89.335
```

7.18.7 References

1. Training a Classifier in PyTorch, available at: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html.

BACK TO TOP

7.19 Lecture 19 - Natural Language Processing

- *References*

7.19.1 19.1 Introduction to NLP

Natural Language Processing (NLP) is a branch of Computer Science (and more broadly, a branch of Artificial Intelligence) that is concerned with providing computers with the ability to understand texts and human language.

Common tasks in NLP include:

- *Text classification* — assign a class label to text based on the topic discussed in the text, e.g., sentiment analysis (positive or negative movie review), spam detection, content filtering (detect abusive content).
- *Text summarization/reading comprehension* — summarize a long input document with a shorter text.
- *Speech recognition* — convert spoken language to text.
- *Machine translation* — convert text in a source language to a target language.
- *Part of Speech (PoS) tagging* — mark up words in text as nouns, verbs, adverbs, etc.
- *Question answering* — output an answer to an input question.
- *Dialog generation* — generate the next reply in a conversation given the history of the conversation.
- *Text generation* — generate text to complete the sentence or to complete the paragraph.

7.19.2 19.2 Preprocessing Text Data

In order to perform operations with text data, they first need to be converted into a numerical representation.

Converting text data into numerical form for processing by ML models typically involves the following steps:

- *Standardization* - remove punctuation, convert the text to lowercase.
- *Tokenization* - break up the text into tokens (e.g., tokens can be individual words, several consecutive words (N-grams), or individual characters).
- *Indexing* - assign a numerical index to each token in the training set (i.e., vocabulary). Modern ML models typically include an additional step - *embedding* which involves assigning a numerical vector to each token (e.g., one-hot encoding and word-embedding are explained in Section 19.5 below).

Text Standardization

Text standardization usually includes some or all of the following steps, depending on the application:

- Remove punctuation marks (such as comma, period) or non-alphabetic characters (@, #, {,]).
- Change all words to lower-case letters, since ML models should consider *Text* and *text* as the same word.

Some NLP tasks can apply additional steps, such as:

- Correct spelling errors or replace abbreviations with full words.
- Remove stop words, such as *for*, *the*, *is*, *to*, *some*, etc.; if the task is text classification, these words are not relevant to the meaning of the text.
- Apply stemming and lemmatization, which transforms words to their base form, such as changing the word *changing* to *change*, or *grilled* to *grill* since they have a common root.

Applying text standardization is helpful for training ML models, because the models do not need to consider *Text* and *text* as two different words, which reduces the requirements for large training datasets. However, depending on the application, text standardization may remove information that can be important for some tasks, and this should always be considered when performing text preprocessing.

Tokenization

Tokenization is breaking up a sequence of text into individual units called *tokens*.

Tokenization can be performed at different levels:

- *Character-level tokenization* - where the text is divided into individual characters, and each character is a token, including letters, digits, punctuation marks, and symbols. One disadvantage of this type of tokenization is that antigrams (words with same letters in different order, such as *silent* and *listen*) can have the same numerical encoding, which can affect the performance of ML models. As well as, character-level tokenization does not capture semantic meaning of words as effectively as word-level tokens. Consequently, it is not widely used in practice.
- *Word-level tokenization* - where each word is a token. This type of tokenization provides a natural representation of input text with the words as building blocks of language, and it is the most commonly used.
- *Subword-level tokenization* - where the words are divided into smaller units (e.g., tokenizing the word "unhappiness" into two tokens "un" + "happiness"). In some languages with complex word structures, subword-level tokenization is more suitable.
- *N-gram tokenization* - where N consecutive words represent a token. For instance, N-grams consisting of two adjacent words are called bigrams, or three words constitute a trigram, etc. N-gram tokens preserve the words order and can potentially capture more information in the text. For instance, for spam filtering using bigram tokens such as *mailing list* or *bank account* may provide more helpful information than using word-level tokens.

For some NLP tasks, tokenization can also be performed at other levels, such as sentence-level tokenizaiton for document segmentation task.

An example of text standardization, word-level tokenization, and indexing is shown in the next figure.

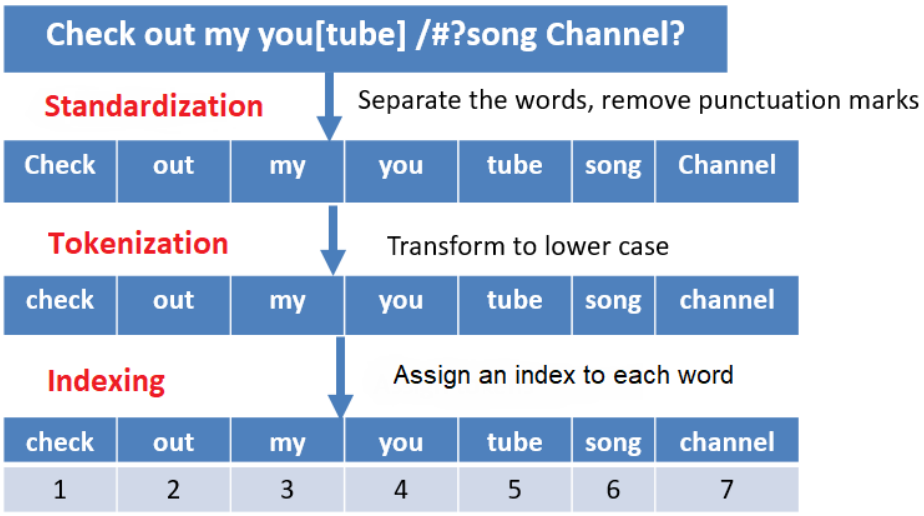


Figure: Text standardization, word-level tokenization, and indexing.

7.19.3 19.3 Text Tokenization

Keras provides a text preprocessing function `Tokenizer` for converting raw text into sequences of tokens. The `Tokenizer` performs text standardization, tokenization, and indexing.

The Kears `Tokenizer` has the following arguments:

- *num_words*: the maximum number of words to keep in the input text. It is better to set a high number if we are not sure, because if we set a number less than the words in the text, some words will not be tokenized.
- *filters*: by default, all punctuations and special characters in the text will be removed. If we want to change that, we can provide a list of punctuations and characters to keep.
- *lower*: can be True or False. By default, it is True, and that means all texts will be converted to lowercase.
- *split*: separator for splitting words. A default separator is a space (`" "`).
- *char_level*: can be True or False. By default, it is False and will perform word-level tokenization. If it is True, the function will perform character-level tokenization.
- *oov_token*: oov stands for Out Of Vocabulary, and it denotes a special token that will replace tokens that are not present in the input text.

19.3.1 Character-level Tokens

To use the `Tokenizer` for character-level tokenization, we need to set `char_level` to `True`. Let's set the number of tokens to 1,000.

Let's apply it to the following sentence by using the method `fit_on_texts()`.

[ ]:
```
import tensorflow as tf
from tensorflow import keras
import numpy as np

# Print the version of tf
print("TensorFlow version:{}".format(tf.__version__))
```

```
TensorFlow version:2.14.0
```

[ ]:
```
# A sample sentence
sentence = ['TensorFlow is a Machine Learning framework']
```

[ ]:
```
from keras.preprocessing.text import Tokenizer

tokenizer = Tokenizer(num_words=1000, char_level=True)

# Fitting tokenizer on sentences
tokenizer.fit_on_texts(sentence)
```

When the `Tokenizer` separates the characters in text, it creates a dictionary that maps each character to an integer index. We can inspect the dictionary by using the attribute `word_index`, although since we have set `char_level` to `True` in this case it is the character index.

Note that the start index is 1. By default, all letters are converted to lowercase. The first token is an empty space `' '`, the second is the letter `'e'`, etc. There are 17 unique characters in the sentence, including the empty space.

[ ]:
```
char_index = tokenizer.word_index
print(char_index)
```

```
{' ': 1, 'e': 2, 'n': 3, 'r': 4, 'a': 5, 'o': 6, 'i': 7, 's': 8, 'f': 9, 'l': 10, 'w':␣
↪11, 'm': 12, 't': 13, 'c': 14, 'h': 15, 'g': 16, 'k': 17}
```

The method `text_to_sequences` outputs the indices for the text. You can check that the word `TensorFlow` has the indices 13, 2, 3, 8, 6, 4, 9, 10, 6, 11, where each index corresponds to the letters listed in `char_index`.

[ ]:
```
print(tokenizer.texts_to_sequences(sentence))
```

```
[[13, 2, 3, 8, 6, 4, 9, 10, 6, 11, 1, 7, 8, 1, 5, 1, 12, 5, 14, 15, 7, 3, 2, 1, 10, 2, 5,
↪ 4, 3, 7, 3, 16, 1, 9, 4, 5, 12, 2, 11, 6, 4, 17]]
```

As we mentioned earlier, character-level tokenization is rarely used, because it does not capture semantic meaning of words as effectively as word-level tokens.

19.3.2 Word-level Tokens

To use the `Tokenizer` for tokenizing words instead of characters, we need to just change the argument `char_level` to `False`, which is the default setting, so we may as well just omit it.

[ ]:
```
# Sample sentences
sentences = ['TensorFlow is a Machine Learning framework.',
             'Keras is a well designed deep learning API!',
             'Keras is built on top of TensorFlow!']
```

After the text is broken down into individual words, the `Tokenizer` builds a *vocabulary* of all words that are found in the input text, and assigns a unique integer index to each word in the vocabulary. We can inspect the words by using again the attribute `word_index`.

[ ]:
```
tokenizer = Tokenizer(num_words=1000)

# Fitting tokenizer on sentences
tokenizer.fit_on_texts(sentences)

word_index = tokenizer.word_index
print(word_index)
```

```
{'is': 1, 'tensorflow': 2, 'a': 3, 'learning': 4, 'keras': 5, 'machine': 6, 'framework':␣
↪7, 'well': 8, 'designed': 9, 'deep': 10, 'api': 11, 'built': 12, 'on': 13, 'top': 14,
↪'of': 15}
```

There are 15 unique words in the above sentences. By default, all punctuations are removed and all letters are converted to lowercase.

The indices for the above three sentences are shown below. For instance, the first list `[2, 1, 3, 6, 4, 7]` represents the first sentence in the text `TensorFlow is a Machine Learning framework`.

[ ]:
```
print(tokenizer.texts_to_sequences(sentences))
```

```
[[2, 1, 3, 6, 4, 7], [5, 1, 3, 8, 9, 10, 4, 11], [5, 1, 12, 13, 14, 15, 2]]
```

Also, `word_counts` can return the number of times each word appears in the sentences.

[ ]:
```
word_counts = tokenizer.word_counts
word_counts
```

```
OrderedDict([('tensorflow', 2),
             ('is', 3),
             ('a', 2),
             ('machine', 1),
             ('learning', 2),
             ('framework', 1),
             ('keras', 2),
             ('well', 1),
             ('designed', 1),
             ('deep', 1),
             ('api', 1),
             ('built', 1),
             ('on', 1),
             ('top', 1),
             ('of', 1)])
```

Out of Vocabulary Words

To handle the case when the Tokenizer is applied to text that contains words that were not present in the original documents, we can define a special token `oov_token`. This token will be used to replace these words that are Out Of Vocabulary (OOV).

In the example below, we set the `oov_token`, which has been assigned the index 1.

[ ]:
```
tokenizer = Tokenizer(num_words=1000, oov_token='Word Out of Vocab')
tokenizer.fit_on_texts(sentences)
word_index = tokenizer.word_index
print(word_index)
```

```
{'Word Out of Vocab': 1, 'is': 2, 'tensorflow': 3, 'a': 4, 'learning': 5, 'keras': 6,
↪'machine': 7, 'framework': 8, 'well': 9, 'designed': 10, 'deep': 11, 'api': 12, 'built
↪': 13, 'on': 14, 'top': 15, 'of': 16}
```

[ ]:
```
# Converting text to sequences
print(tokenizer.texts_to_sequences(sentences))
```

```
[[3, 2, 4, 7, 5, 8], [6, 2, 4, 9, 10, 11, 5, 12], [6, 2, 13, 14, 15, 16, 3]]
```

Next, if we pass text with new words that the tokenizer was not fit to, the new words will be replaced with the `oov_token`.

[ ]:
```
new_sentences = ['I like TensorFlow', # 'I' and 'like' are new words
                 'Keras is a superb deep learning API'] # 'superb' is a new word

print(tokenizer.texts_to_sequences(new_sentences))
```

```
[[1, 1, 3], [6, 2, 4, 1, 11, 5, 12]]
```

And also, if we work with a large dataset that contains many documents, we can limit the number of words in the vocabulary to 20,000 or 30,000, and consider the rare words as out-of-vocabulary words. This can reduce the input space of the model, by ignoring those words that are present only once or twice in the large database.

19.3.3 Padding Word Sequences

Most machine learning models require the input samples to have the same length/size. In Keras, the function `pad_sequences()` can be used to pad the text sequences with predefined values, so that they have the same length.

The function `pad_sequences()` accepts the following arguments:

- *sequence*: a list of integer indices (i.e., tokenized text).
- *maxlen*: maximum length of all sequences; if not provided, sequences will be padded to the length of the longest sequence.
- *padding*: 'pre' (default) or 'post', whether to pad before the sequence or after the sequence.
- *truncating*: 'pre' (default) or post', whether to remove the values from sequences larger than maxlen at the beginning or at the end of the sequences.
- *value*: a float or a string to use as a padding value. By default, the sequences are padded with 0.

[ ]:
```
tokenized_sentences = tokenizer.texts_to_sequences(sentences)

print(tokenized_sentences)
```

```
[[3, 2, 4, 7, 5, 8], [6, 2, 4, 9, 10, 11, 5, 12], [6, 2, 13, 14, 15, 16, 3]]
```

The next cell shows the above sequences pre-padded with 0 to sequences with length 10.

[ ]:
```
from keras.preprocessing.sequence import pad_sequences

padded_sequences = pad_sequences(tokenized_sentences, maxlen=10)

print(padded_sequences)
```

```
[[ 0  0  0  0  3  2  4  7  5  8]
 [ 0  0  6  2  4  9 10 11  5 12]
 [ 0  0  0  6  2 13 14 15 16  3]]
```

7.19.4 19.4 Representation of Groups of Words

Representation of groups of words in Machine Learning models for text processing includes two categories of approaches:

- **Set models** approach, the text is represented as unordered collection of words. Such approaches include *bag-of-words* models.
- **Sequence models** approach, where the text is represented as ordered sequences of words. These methods preserve the order of the words in the text. Representatives of these approaches are Recurrent Neural Networks, and Transformer Networks.

The order of words in natural language is not necessarily fixed, and sentences with different orders of the words can have the same meaning. Also, different languages use different ways to order the words. As a result, defining the order of the words in text in NLP tasks is not straightforward.

Bag-of-Words Models

Bag-of-words models discard the information about the order of the words, where the term *bag* implies that the structure of the text is lost. A depiction of a bag-of-words is shown below, where the initial text is separated into word-level tokens, and a bag is created from all words in the text. Also, instead of individual words, these models often employ N-gram representations. This type of models typically consider the frequency of occurrence of each word in the training data, and a classifier is trained by using the word counts as inputs.

For instance, to create a spam filtering classifier, two bags-of-words can be created from the words in spam and non-spam emails. Presumably, the spam bag will contain trigger words (such as cheap, buy, stock) more frequently than the bag with words from non-spam emails. A classifier will be trained using the two bags-of-words and learn to differentiate trigger words from regular words. After the training, the classifier will analyze the words in new unseen messages, and predict the probability that these words belong to the spam or non-spam bag-of-words.

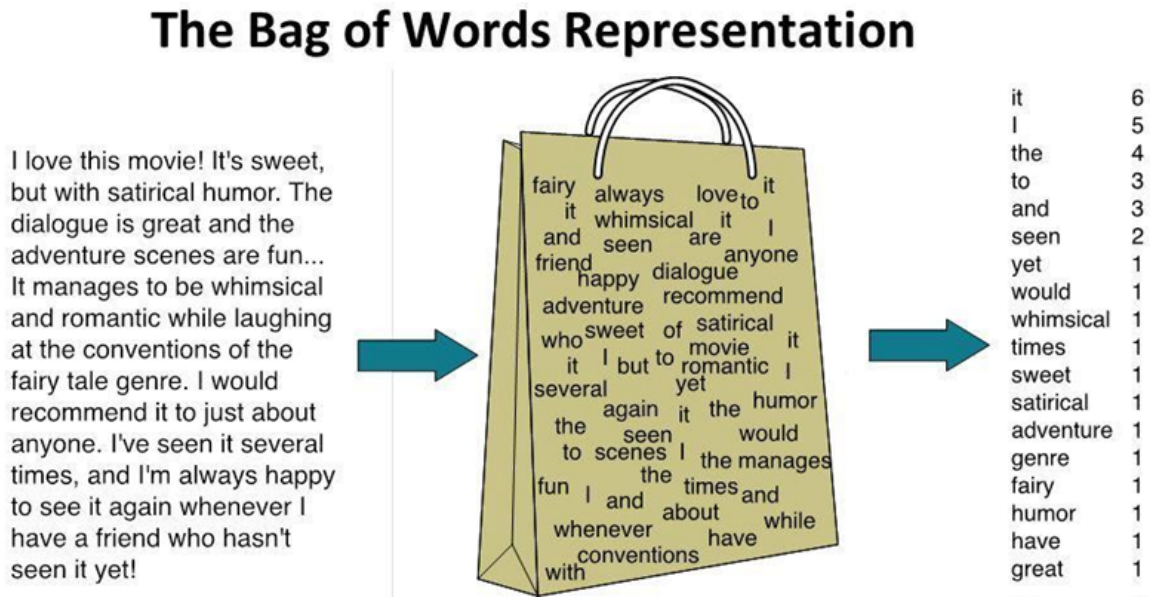


Figure: Bag-of-words representation.

The early applications of machine learning in NLP relied on bag-of-words models. Modern applications, especially those related to large language models, rely predominantly on sequence models. Before 2018, Recurrent Neural Networks were the preferred models for NLP applications. In recent years, Transformer Networks have replaced Recurrent Neural Networks as more powerful models for NLP tasks.

7.19.5 19.5 Sequence Models Approach

Sequence models process the entire text sequence at once, which allows preserving the order of words in the input text. Typical implementation of sequence models includes the steps of representing the words in text data with integer indices, mapping the integers to vector representations, and passing the vectors to a machine learning model, where the layers in the model will account for the ordering of input vectors.

The input vectors to sequence models can be in the form of:

- One-hot word vector representation, or
- Word embeddings representation.

One-Hot Word Vector Representation

One-hot word vector representation is similar to encoding categorical features with one-hot encoding matrix. That is, the index for each word is converted to one-hot vector, having `1 (hot)` for that word and `0 (cold)` for all other words. An example is shown in the left-hand figure, where we created a zero vector with length of 4, and assigned 1 for the index that corresponds to every word. Another example if shown in the right-hand figure.

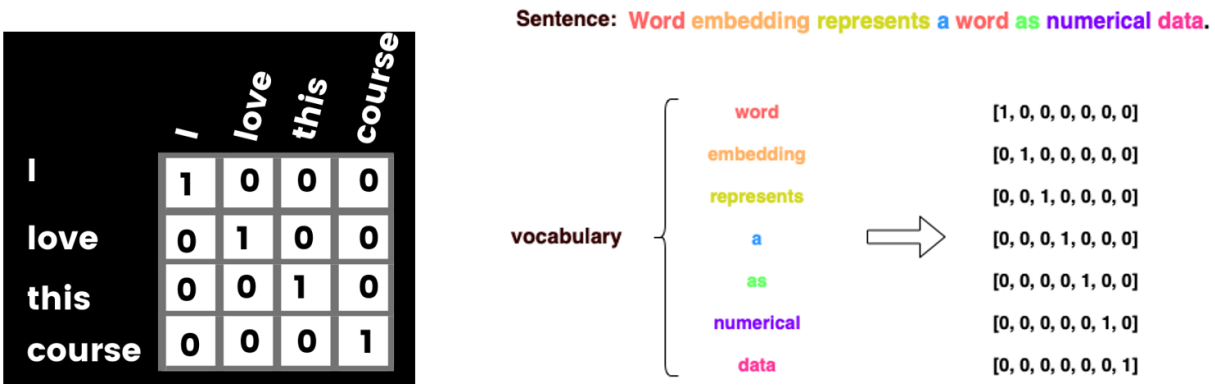


Figure: One-hot word vector encoding.

One-hot word vector representation is not an efficient way to represent text, because for large text datasets the input vectors can become quite large. For instance, a training set with 20,000 words will need to use one-hot vectors of size 20,000 to represent each word, and this results in slow training, as well as this type of word representation takes a lot of memory space.

Using word embeddings is more efficient, since the vectors for word representation are much smaller than the size of the vocabulary, and more importantly, embedding vectors can capture important semantic meaning of the words. Hence, most modern NLP models rely on word embeddings for word representating words in text.

19.5.1 Word Embeddings

Word embeddings representation is used to convert each word into a vector (also referred to as embedding vector), in such as way that the vectors of words that have similar semantic meaning have close spatial positions in the embeddings space.

The embeddings space consists of the set of vectors, where each word in the vocabulary is represented with one vector. For calculating the distance between the vectors in the embeddings space, typically the cosine similarity is used as a distance metric. For two vectors u and v, *cosine similarity* is calculated as the dot (scalar, inner) product of the vectors divided by the norm of the vectors, i.e., $\frac{u \cdot v}{||u|| \cdot ||v||}$.

Typical vectors for representing word embeddings have between 256 to 1,024 dimensions. For instance, the following figure shows the embedding vector for the word 'work'. The embedding vector has many values, and each value represents some aspect of the meaning of that word.

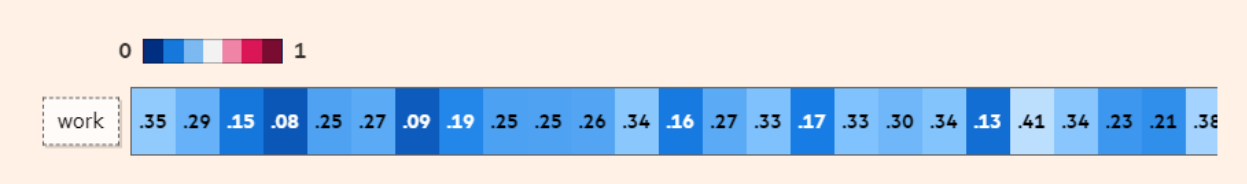


Figure: Embedding vector for the word 'work'. Source: link

The embedding vectors of words that have similar meanings are also similar. In the following figure, we can see that the embedding vectors of the words 'football' and 'soccer' are more similar to each other, than the embedding vectors of the words 'sea' or 'we'.

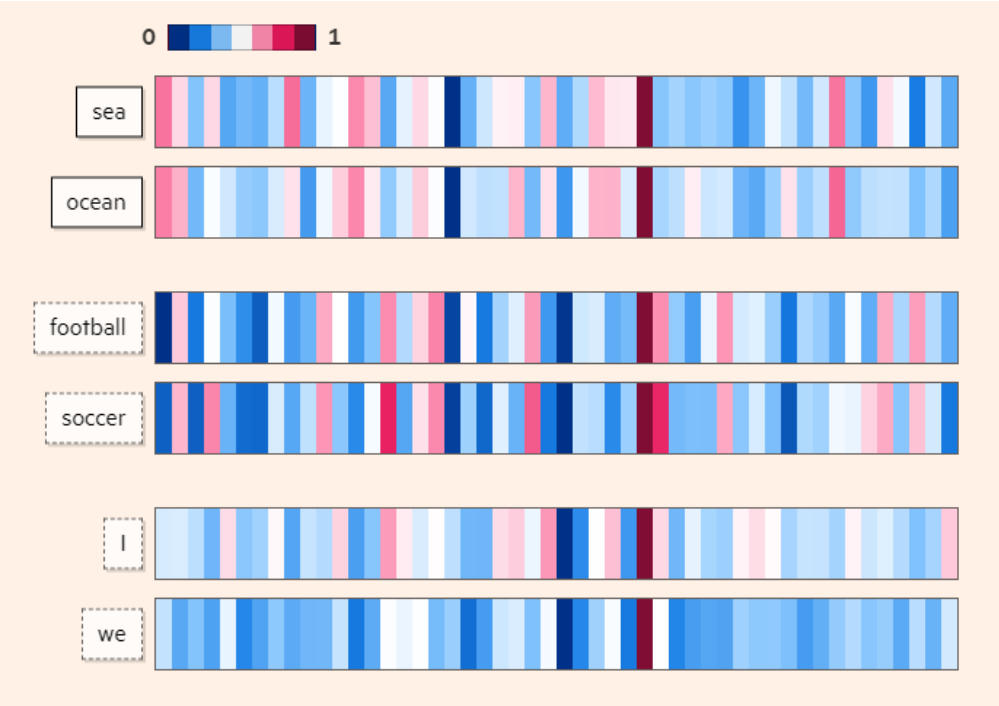


Figure: Embedding vectors for words with similar meanings are also similar. Source: link

A simple example of word embeddings space is shown below, where similar words are positioned closer to each other. Therefore, the spatial distance between the vectors is dependent on the semantic meaning of the words.

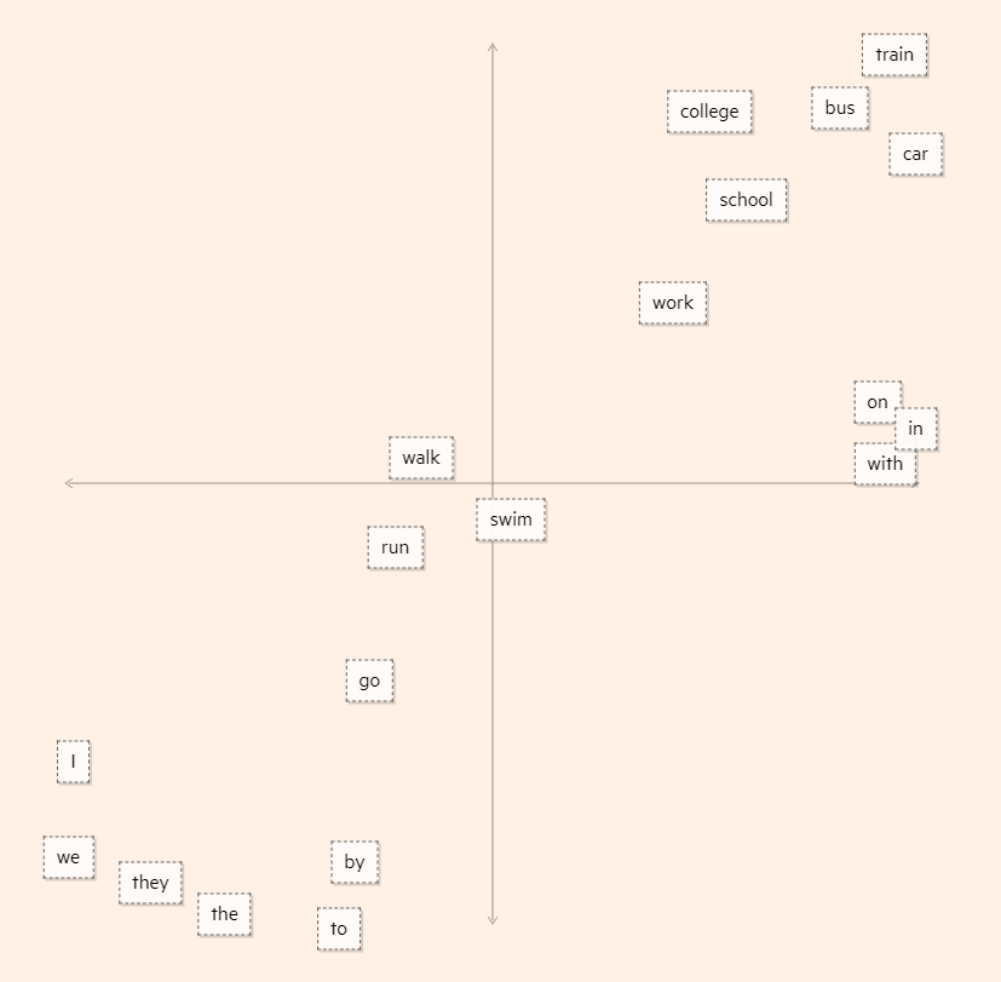


Figure: Word embeddings space. Source: link

Two popular methods for generating word embeddings are *word2vec* and *Glove*. These methods use neural networks to learn embedding vectors from a large corpus of text. The resulting vectors learned by these techniques can be imported as pretrained word embeddings and applied to downstream tasks with smaller training datasets.

For example, the website Embedding Projector provides visualizations of word embeddings, and for an entered word displays other words that are adjacent in the embeddings space.

To demonstrate the use of word embeddings with Keras, we will implement it for a sentiment analysis task, to classify movie reviews using the IMDB Reviews dataset.

Loading the IMDB Reviews Dataset

IMDB Reviews Dataset can be downloaded from the built-in datasets in Keras. There are 25,000 samples of movie reviews for training and 25,000 samples for validation. Setting `max_features` to 20,000 means we are only considering the first 20,000 words and the rest of the words will have the out-of-vocabulary token. Each movie review has a positive or negative label.

The training and validation datasets will be loaded as lists with 25,000 elements.

[ ]:
```
max_features = 20000
```

(continues on next page)

(continued from previous page)

```
(train_data, train_labels), (val_data, val_labels) = keras.datasets.imdb.load_data(num_
↪words=max_features)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.
↪npz
17464789/17464789 [==============================] - 0s 0us/step
```

[ ]:
```
print(len(train_data))
print(len(val_data))
```

```
25000
25000
```

Displayed below is one example of a movie review. It is a list of indices, it contains 141 words, and as we can see the words in the dataset are already converted to integer indices.

[ ]:
```
# Display the third movie review
print('Number of words in the third review', len(train_data[2]))
print(train_data[2])
```

```
Number of words in the third review 141
[1, 14, 47, 8, 30, 31, 7, 4, 249, 108, 7, 4, 5974, 54, 61, 369, 13, 71, 149, 14, 22, 112,
↪ 4, 2401, 311, 12, 16, 3711, 33, 75, 43, 1829, 296, 4, 86, 320, 35, 534, 19, 263, 4821,
↪ 1301, 4, 1873, 33, 89, 78, 12, 66, 16, 4, 360, 7, 4, 58, 316, 334, 11, 4, 1716, 43,␣
↪645, 662, 8, 257, 85, 1200, 42, 1228, 2578, 83, 68, 3912, 15, 36, 165, 1539, 278, 36,␣
↪69, 2, 780, 8, 106, 14, 6905, 1338, 18, 6, 22, 12, 215, 28, 610, 40, 6, 87, 326, 23,␣
↪2300, 21, 23, 22, 12, 272, 40, 57, 31, 11, 4, 22, 47, 6, 2307, 51, 9, 170, 23, 595,␣
↪116, 595, 1352, 13, 191, 79, 638, 89, 2, 14, 9, 8, 106, 607, 624, 35, 534, 6, 227, 7,␣
↪129, 113]
```

[ ]:
```
# Display the first 10 train labels
train_labels[:10]
```

```
array([1, 0, 0, 1, 0, 0, 1, 0, 1, 0])
```

Preparing the Dataset

Let's pad the data using the `pad_sequences` function in Keras. Setting `maxlen` indicates to use the first 200 words in each movie review, and ignore the rest. Most movie reviews in the dataset are shorter than 200 words, however for those that are longer than 200 words some information will be lost. That is a tradeoff between computational expense and model performance.

We can see in the next cell that for the third review, which has a length of 141 words, the first 59 words are now 0, and the length is 200.

[ ]:
```
train_data = pad_sequences(train_data, maxlen=200)
val_data = pad_sequences(val_data, maxlen=200)
```

[ ]:
```
# Display the third movie review
print('Shape of the third padded review:', train_data.shape, '\n')
print(train_data[2])
```

```
Shape of the third padded review: (25000, 200)

[   0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    1   14   47    8   30   31    7    4  249  108    7
    4 5974   54   61  369   13   71  149   14   22  112    4 2401  311
   12   16 3711   33   75   43 1829  296    4   86  320   35  534   19
  263 4821 1301    4 1873   33   89   78   12   66   16    4  360    7
    4   58  316  334   11    4 1716   43  645  662    8  257   85 1200
   42 1228 2578   83   68 3912   15   36  165 1539  278   36   69    2
  780    8  106   14 6905 1338   18    6   22   12  215   28  610   40
    6   87  326   23 2300   21   23   22   12  272   40   57   31   11
    4   22   47    6 2307   51    9  170   23  595  116  595 1352   13
  191   79  638   89    2   14    9    8  106  607  624   35  534    6
  227    7  129  113]
```

Embedding Layer in Keras

Keras has `Embedding` layer, which we will use to project the input tokens into vectors in an embedding space. The `Embedding` layer requires at the minimum to specify the number of possible tokens in the data sequences, and the dimensionality of the vectors in the embeddings space. The layer takes integer indices as inputs, and outputs a feature vector. It can be considered as a look-up table, which maps an embedding vector to each integer index.

To understand how the Embedding layer works, let's consider a dataset with the maximum number of words set to 100, and out aim is to represent the words with 5-dimensional vectors. In the cell below, the Embedding layer assigned random values to the list of indices 1, 2, and 3, and we can see that to each index a 5-dimensional vector is assigned. However, the embedding vectors are trainable, and when we include the Embedding layer in a model, as we train the model, words that are similar will get closer in the embeddings space.

[ ]:
```
from keras.layers import Embedding

# embedding layer: represent a dataset with a vocabulary of 100 words with 5 dimensional␣
↪vectors
embedding_layer = Embedding(input_dim=100, output_dim=5)

embed_integers = embedding_layer(tf.constant([1, 2, 3]))

embed_integers.numpy()
```

```
array([[ 0.04004519,  0.0458275 , -0.04573163, -0.02836338,  0.00029951],
       [-0.03247341,  0.00142381, -0.00083622, -0.04686186, -0.02296721],
       [ 0.04272026,  0.00845009, -0.04817088, -0.04544125, -0.03160852]],
      dtype=float32)
```

Define, Compile, and Train the Model

Next, we will define a model that uses an `Embedding` layer to project the words in input sequences into 8-dimensional vectors. These vectors will be further processed through dense layers, and the last layer will predict the label of movie reviews. There are two labels: positive and negative movie review, therefore this is a binary classification problem.

[ ]:
```
from keras.layers import Flatten, Dense, Dropout
from keras import Sequential

embedding_dim = 8

# Create a model
model = Sequential([
       Embedding(input_dim=max_features, output_dim=embedding_dim, input_length=200),
       Flatten(),
       Dense(32, activation='relu'),
       Dropout(0.5),
       Dense(1, activation='sigmoid')
])
```

We will compile the model with `binary_crossentropy` loss (two labels: positive and negative review) and `adam` optimizer.

[ ]:
```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Before training the model, we can see the model summary.

[ ]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 200, 8)            160000

 flatten (Flatten)           (None, 1600)              0

 dense (Dense)               (None, 32)                51232

 dropout (Dropout)           (None, 32)                0

 dense_1 (Dense)             (None, 1)                 33

=================================================================
Total params: 211265 (825.25 KB)
Trainable params: 211265 (825.25 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

[ ]:
```
history = model.fit(train_data, train_labels, validation_data = (val_data, val_labels),␣
↪epochs=3)
```

```
Epoch 1/3
782/782 [==============================] - 75s 85ms/step - loss: 0.4589 - accuracy: 0.
↪7602 - val_loss: 0.2947 - val_accuracy: 0.8741
```

(continues on next page)

(continued from previous page)

```
Epoch 2/3
782/782 [==============================] - 18s 23ms/step - loss: 0.1971 - accuracy: 0.
↪9291 - val_loss: 0.3094 - val_accuracy: 0.8733
Epoch 3/3
782/782 [==============================] - 13s 17ms/step - loss: 0.0863 - accuracy: 0.
↪9736 - val_loss: 0.3993 - val_accuracy: 0.8626
```

19.5.2 Using TextVectorization Layer

Keras also provides another way to preprocess text by using a `TextVectorization` layer.

This layer performs the following preprocessing steps:

- Standardize text by removing punctuations and lowering the text case.
- Split sentences into individual tokens.
- Convert the tokens into a numerical representation.

The arguments in `TextVectorization` layer are:

- *max_tokens*: maximum number of tokens in the vocabulary, where vocabulary is comprised of unique text units (words) in the data. E.g., if `max_tokens=1000` the layer will only consider the 1000 most frequent tokens from the input text data when building the vocabulary.
- *standardize*: denotes the standardization specifics to be applied to input data; by default, it is `lower_and_strip_punctuation` meaning to convert to lowercase and remove punctuations.
- *split*: denotes what will be considered while splitting the input text; by default it is whitespace `" "`.
- *output_sequence_length*: the length to which the sequences will be padded (if shorter than the length) or truncated (if longer than the length).

[ ]:
```
from keras.layers import TextVectorization
```

[ ]:
```
# Sample sentences
sentences = ['TensorFlow is a deep learning library!',
             'Is TensorFlow powered by Keras API?']
```

[ ]:
```
text_vect_layer = TextVectorization(max_tokens=1000, output_sequence_length=10)
```

The `adapt()` method is used to fit the sentences to the TextVectorization layer. The `adapt()` method will create a vocabulary of the most frequent tokens, and it will create a mapping from tokens to integer indices that will be used later for converting text into a numerical representation.

[ ]:
```
text_vect_layer.adapt(sentences)
```

Let's pass a sample sentence to inspect the output.

[ ]:
```
sample_sentence = 'Tensorflow is a machine learning framework!'

vectorized_sentence = text_vect_layer([sample_sentence])
```

[ ]:
```
print('Orginal sentence:', sample_sentence)
print('Vectorized sentence:', vectorized_sentence)
```

```
Orginal sentence: Tensorflow is a machine learning framework!
Vectorized sentence: tf.Tensor([[ 2  3 11  1  6  1  0  0  0  0]], shape=(1, 10),␣
↪dtype=int64)
```

Since the words `'machine'` and `'framework'` were not part of the `sentences` that we passed to the layer, they are both represented by 1 in the vectorized output, since the index 1 is reserved for words that are out of vocabulary(`oov_token`).

The output is padded with 0, and the length of the output sequence size is 10.

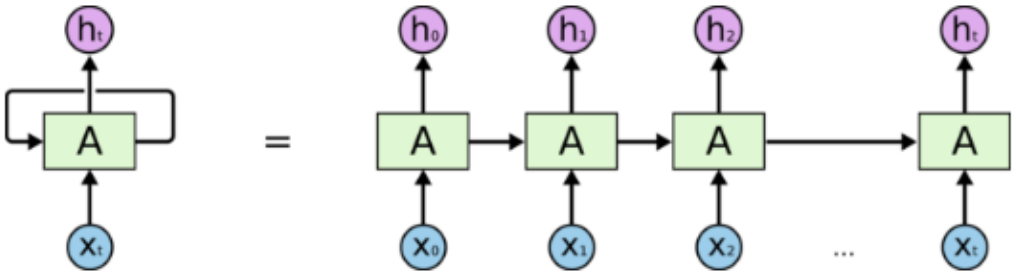
The `TextVectorization` layer performs all required text preprocessing steps at once, and another advantage of this layer is that it can be used inside a model.

19.5.3 Sequence Modeling with Recurrent Neural Networks

Recurrent Neural Networks (RNN) is a neural network architecture that is designed for handling sequential data. Examples of sequential data are time-series, texts (sequence of words or characters), audio (sequence of sound waves), etc.

Working with sequential data requires to preserve the sequence of the information flow in the data. For example, given the sentence `Today, I took my cat for a [....]`, to predict the next word, there should be a way to capture and preserve the flow from the beginning to the end of the sequence.

In conventional feedforward networks (such as networks composed of fully-connected or convolutional layers), the information flows from the input layer to the output layer. Conversely, in RNNs, there is a feedback loop at each time step, which creates the *recurrence*. This is shown in the next figure, where at each time step of the RNN model, an input (e.g., word) is processed, then in the next step the succeeding word is processed based on the information from the previous word, etc. This way, the network can learn dependencies between words that are not adjacent.



An unrolled recurrent neural network.

Figure: Recurrent Neural Network.

There are three major types of RNN layers: conventional (a.k.a. basic, simple, vanilla) RNN, LSTM (Long Short-Term Memory), and GRU (Gated Recurrent Units). They are implemented in Keras and PyTorch, and can be conveniently imported and used for creating models. In Keras, the conventional (basic) RNN is called SimpleRNN, and LSTM and GRU are called as they are written.

While SimpleRNN has difficulty in handling long sequences, LSTM and GRU have the ability to store and preserve long-term dependencies over many time steps. Consequently, SimpleRNNs are rarely used at present.

Both LSTM and GRU layers use multiple gates to control to flow of information between the time steps. For instance, LSTM layers include an input gate and an output gate to control the input and output information for each time step, a forget gate that removes irrelevant information, and a memory cell that saves important information.

Next, we will apply an RNN model with LSTM layers for classification of text data.

Loading the Data

We are going to use the `ag_news_subset` dataset that is available in TensorFlow datasets. AG is a collection of news articles gathered from more than 2,000 news sources. The news articles are classified into 4 classes: World(0), Sports(1), Business(2), and Sci/Tech(3). The total number of training samples is 120,000 and testing 7,600.

Let's get the dataset from TensorFlow datasets. In the load function, `with_info=True` will return various information about the dataset (as shown in the next cells), and `as_supervised=True` indicates that the data will be loaded as 2-element tuples consisting of (input, target) pairs.

[ ]:
```
import tensorflow_datasets as tfds
import pandas as pd
```

[ ]:
```
(train_data, val_data), info = tfds.load('ag_news_subset:1.0.0', #version 1.0.0
                                         split=['train', 'test'],
                                         with_info=True,
                                         as_supervised=True)
```

```
Downloading and preparing dataset 11.24 MiB (download: 11.24 MiB, generated: 35.79 MiB,␣
↪total: 47.03 MiB) to /root/tensorflow_datasets/ag_news_subset/1.0.0...
```

```
Dl Completed...: 0 url [00:00, ? url/s]
```

```
Dl Size...: 0 MiB [00:00, ? MiB/s]
```

```
Extraction completed...: 0 file [00:00, ? file/s]
```

```
Generating splits...:   0%|          | 0/2 [00:00<?, ? splits/s]
```

```
Generating train examples...:   0%|          | 0/120000 [00:00<?, ? examples/s]
```

```
Shuffling /root/tensorflow_datasets/ag_news_subset/1.0.0.incomplete665NZN/ag_news_subset-
↪train.tfrecord*...:  ...
```

```
Generating test examples...:   0%|          | 0/7600 [00:00<?, ? examples/s]
```

```
Shuffling /root/tensorflow_datasets/ag_news_subset/1.0.0.incomplete665NZN/ag_news_subset-
↪test.tfrecord*...:   ...
```

```
Dataset ag_news_subset downloaded and prepared to /root/tensorflow_datasets/ag_news_
↪subset/1.0.0. Subsequent calls will reuse this data.
```

We can use `info` to check basic information about the dataset.

[ ]:
```
# Displaying the classes
class_names = info.features['label'].names
print(class_names)
```

```
['World', 'Sports', 'Business', 'Sci/Tech']
```

[ ]:
```
print('Number of training samples:', info.splits['train'].num_examples)
print('Number of validation samples:', info.splits['test'].num_examples)
```

```
Number of training samples: 120000
Number of validation samples: 7600
```

We can use `tfds.as_dataframe` to display the first 10 news articles as Pandas DataFrame.

[ ]:
```
news_df = tfds.as_dataframe(train_data.take(10), info)

news_df
```

```
                                         description  label
0  b'AMD #39;s new dual-core Opteron chip is desi...      3
1  b'Reuters - Major League Baseball\\Monday anno...      1
2  b'President Bush #39;s  quot;revenue-neutral q...      2
3  b'Britain will run out of leading scientists u...      3
4  b'London, England (Sports Network) - England m...      1
5  b'TOKYO - Sony Corp. is banking on the \\$3 bi...      0
6  b'Giant pandas may well prefer bamboo to lapto...      3
7  b'VILNIUS, Lithuania - Lithuania #39;s main pa...      0
8  b'Witnesses in the trial of a US soldier charg...      0
9  b'Dan Olsen of Ponte Vedra Beach, Fla., shot a...      1
```

The columns in the DataFrame are `description` and `label`.

[ ]:
```
news_df.columns
```

```
Index(['description', 'label'], dtype='object')
```

Now that we understand the data, let's prepare it before we can use LSTMs to classify the news.

Preparing the Data

First, we will shuffle and batch the training data. For the validation data, we don't shuffle, we only batch it.

The `buffer_size` below limits the number of data points to be shuffled to 1,000. This can be useful when working with large datasets, that can not fit in the memory.

The `prefetch()` function below is only added for optimizing the performance, and while the model is being trained, it will prefetch batches for validation.

[ ]:
```
buffer_size = 1000
batch_size = 32

train_data = train_data.shuffle(buffer_size)
train_data = train_data.batch(batch_size).prefetch(1)
val_data = val_data.batch(batch_size).prefetch(1)
```

To convert the text data into tokens, in this case we will use the TextVectorizer layer in Keras.

[ ]:
```
max_features = 20000

text_vectorizer = TextVectorization(max_tokens=max_features)
```

Next, we will apply the `adapt()` method to preprocess the training data. Since the data was loaded as (input, label) tuples, the `lambda` function applies the vectorizer only on the input features (in the column `description`), and not on the `labels`. This is, the lambda function that takes two arguments, `description` and `label`, and returns only the `description`. The `map()` method that is called on the `train_data` applies the text_vectorizer to each data instance in the training dataset.

[ ]:
```
text_vectorizer.adapt(train_data.map(lambda description, label: description))
```

Let's pass two news articles to `text_vectorizer`. The vectorized sequences will be padded to the sentence with the maximum length, but if we want to have fixed size of padded sequences, we can set the `output_sequence_length` to another value in the layer initialization.

[ ]:
```
sample_news = ['This weekend there is a sport match between Man U and Fc Barcelona',
               'Tesla has unveiled its humanoid robot that appeared dancing during the␣
↪show!']
```

[ ]:
```
vectorized_news = text_vectorizer(sample_news)
vectorized_news.numpy()
```

```
array([[   40,   491,   185,    16,     3,  1559,   560,   163,   362,
        13418,     7,  7381,  2517],
       [    1,    20,   878,    14,     1,  4663,    10,  1249, 11657,
          159,     2,   541,     0]])
```

Note that the second sentence was padded with 0. Also the words `Tesla` and `humanoid` have an index of 1 because they were not a part of the training data.

Creating and Training the Model

We are going to create a Keras model that takes the tokenized text as input and outputs the class of the news articles.

The model has the following layers:

- `TextVectorization layer` for converting input texts into tokens.
- `Embedding layer` for representing the tokens with trainable embedding vectors. Because the embedding vectors are trainable, words that have similar semantic meaning be represented by vectors that are close in the embeddings space.
- `LSTM layer` for processing the sequences. The layer is wrapped into a Bidirectional layer, which will process the sequences from both directions (forward and backward), i.e., one LSTM layer will process the sequences forward, another layer will process the sequences backward, and the outputs of the two LSTMs will be combined.
- `Dense layer` for classification purpose.

[ ]:
```
input_dim = len(text_vectorizer.get_vocabulary())
input_dim
```

```
20000
```

[ ]:
```
from keras.layers import Bidirectional, LSTM

model = Sequential([
    text_vectorizer,
    Embedding(input_dim=input_dim, output_dim=64),
    Bidirectional(LSTM(64)),
    Dense(64, activation='relu'),
    Dense(4, activation='softmax')
])
```

[ ]:
```
# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=[
↪'accuracy'])
```

[ ]:
```
# Train the model
history = model.fit(train_data, epochs=5, validation_data=val_data)
```

```
Epoch 1/5
3750/3750 [==============================] - 139s 33ms/step - loss: 0.3386 - accuracy: 0.
↪8812 - val_loss: 0.2779 - val_accuracy: 0.9050
Epoch 2/5
3750/3750 [==============================] - 92s 24ms/step - loss: 0.2069 - accuracy: 0.
↪9286 - val_loss: 0.2938 - val_accuracy: 0.9063
Epoch 3/5
3750/3750 [==============================] - 91s 24ms/step - loss: 0.1426 - accuracy: 0.
↪9495 - val_loss: 0.3396 - val_accuracy: 0.9038
Epoch 4/5
3750/3750 [==============================] - 91s 24ms/step - loss: 0.0900 - accuracy: 0.
↪9676 - val_loss: 0.4485 - val_accuracy: 0.8954
Epoch 5/5
3750/3750 [==============================] - 90s 24ms/step - loss: 0.0545 - accuracy: 0.
↪9803 - val_loss: 0.5474 - val_accuracy: 0.8957
```

[ ]:
```
# Predicting the class of new news articles
sample_news_1 = ['The self driving car company Tesla has unveiled its humanoid robot␣
↪that appeared dancing during the show!']

# make predictions on the sample_news 1
predictions_1 = model.predict(sample_news_1)
print(predictions_1)
```

```
1/1 [==============================] - 3s 3s/step
[[0.00460763 0.01687396 0.0096032  0.96891516]]
```

The model correctly predicted that the news article is related to tech or science.

[ ]:
```
# find the index of the predicted class
predicted_class_1 = np.argmax(predictions_1)

print('Predicted class:', predicted_class_1)
print('Predicted class name:', class_names[predicted_class_1])
```

```
Predicted class: 3
Predicted class name: Sci/Tech
```

One more example is provided in the next cell.

[ ]:
```
# Predicting the class of new news
sample_news_2 = ['This weekend there is a match between two big footbal teams in the␣
↪national league']

predictions_2 = model.predict(sample_news_2)

predicted_class_2 = np.argmax(predictions_2)

print('Predicted class:', predicted_class_2)
print('Predicted class name:', class_names[predicted_class_2])
```

```
1/1 [==============================] - 0s 36ms/step
Predicted class: 1
Predicted class name: Sports
```

7.19.6 References

1. Complete Machine Learning Package, Jean de Dieu Nyandwi, available at: https://github.com/Nyandwi/machine_learning_complete.
2. Deep Learning with Python, Francois Chollet, Second Edition, Manning Publications, 2021.

BACK TO TOP

7.20 Lecture 20 - Transformer Networks

7.20.1 20.1 Introduction to Transformers

Transformer Neural Networks, or simply **Transformers**, is a neural network architecture introduced in 2017 in the now-famous paper "Attention is all you need". The title refers to the attention mechanism, which forms the basis for data processing with Transformers.

Transformer Networks have been the predominant type of Deep Learning models for NLP in recent years. They replaced Recurrent Neural Networks in all NLP tasks, and also, all Large Language Models employ the Transformer Network architecture. As well as, Transformer Networks were recently adapted for other tasks and have outperformed other Machine Learning models for image processing and video processing tasks, protein and DNA sequence prediction, time-series data processing, and have been used for reinforcement learning tasks. Consequently, Transformers are currently the most important Neural Network architecture.

7.20.2 20.2 Self-attention Mechanism

Self-attention in NNs is a mechanism that forces a model to attend to portions of the data when making predictions. For instance, in NLP, self-attention mechanism is used to identify words in sentences that have significance for a given query word in the sentence. That is, the model should pay more attention to some words in sentences, and less attention to other words in sentences that are less relevant for a given task.

In the following two sentences, in the left subfigure the word "it" refers to "street", while in the right subfigure the word "it" refers to "animal". Understanding the relationships between the words in such sentences has been challenging with traditional NLP approaches. Transformers use the self-attention mechanism to model the relationships between all words in a sentence, and assign weights to other words in sentences based on their importance. In the left subfigure, the mechanism estimated that the **query word** "it" is most related to the word "street", but the word "it" is also somewhat related to the words "The" and "animal. These words are referred to as **key words** for the query word "it".The intensity of the lines connecting the words, as well as the intensity of the blue color, signifies the attention scores (i.e., weights). The wider and bluer the lines, the higher the attention scores between two words are.

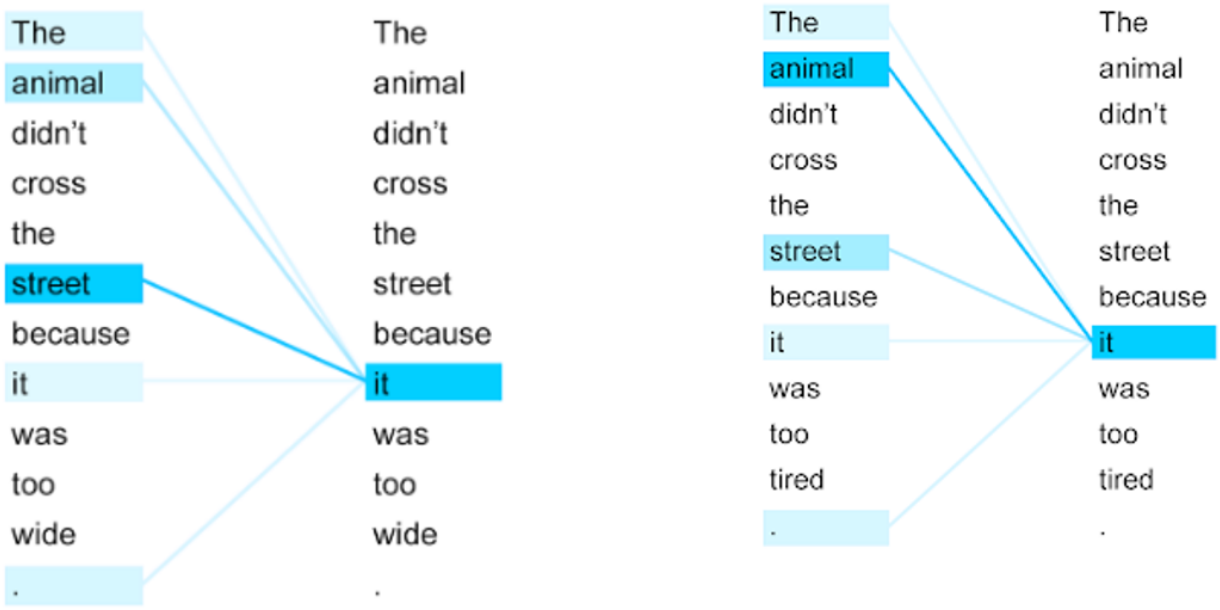


Figure: Attention to words in sentences.

Specifically, Transformer Network compares each word to every other word in the sentence, and calculates attention scores. This is shown in the next figure, where for example, the word "caves" has the highest **attention scores** for the words "glacier" and "formed". The attention scores are calculated as the dot (i.e., inner) product of the input representations of two words. That is, for each Query word Q and Key word K, the attention score is $Q \cdot K$.

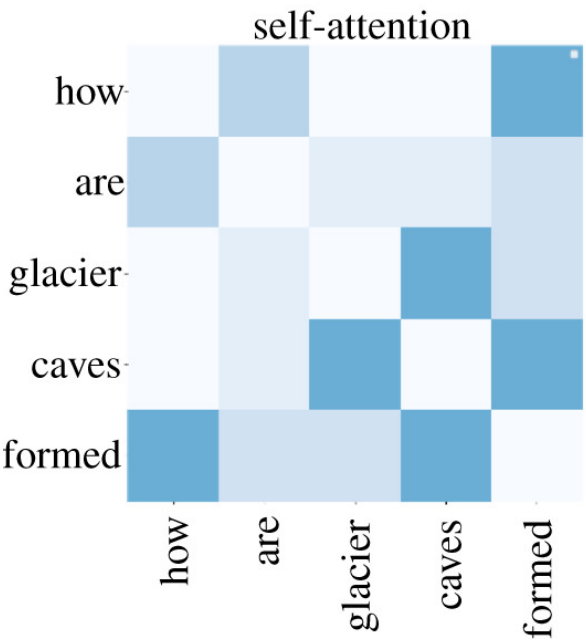


Figure: Attention scores.

Transformers employ word embeddings for representing the individual words in text sequences (where each text sequence can have one or several sentences). Recall from the previous lecture that **word embeddings** are vector representations of words, such that the vectors of words that have similar semantic meaning have close spatial positions in the embeddings space. Therefore, the attention scores are dot products of the embedding vectors for each pair of words in sentences.

The obtained attention scores for each word are then first scaled (by dividing the values by $\sqrt{d}$) and afterward are normalized to be in the [0,1] range (by applying a softmax function). That is, the attention scores are calculated as $a_{ij} = softmax(\frac{Q_i \cdot K_j}{\sqrt{d}})$, where d is the dimensionality of the embedding vectors. Scaling the values by $\sqrt{d}$ is helpful for improving the flow of the gradients during training. The resulting scaled and normalized attention scores are then multiplied with the initial representation of the words, which in the self-attention module is referred to as **value** or V.

This is shown in the next figure. The left subfigure shows the attention scores calculated as product of the input representations of the words Q and K, which are afterwards multiplied with the input representation V to obtain the output of the module. Note that for text classification, all three terms Query, Key, and Value are the same input representation of the words in sentences. However, the original Transformer was developed for machine translation, where the words in the target language are queries, and the words in the source language are pairs of keys and values. This terminology is also related to search engines, which compare queries to keys, and return values (e.g., the user submits a query, the search engine identifies key words within the query to search for, and it returns the results of the search as values). Self-attention works in a similar way, where each query word is matched to other key words, and a weighted value is returned.

The right subfigure below shows how self-attention is implemented in Transformer Networks. Namely, `Matmul` stands for a matrix multiplication layer which calculates the dot product $Q \cdot K$, which is afterwards scaled by $\sqrt{d}$, then there is an optional masking layer , and afterward the final attention scores are obtained by passing it through a `Softmax` layer to obtain $softmax(\frac{Q_i \cdot K_j}{\sqrt{d}})$. Finally, the attention scores are multiplied with V via another matrix multiplication layer `Matmul` to calculate the output of the self-attention module. The optional masking layer can be used for two purposes: (a) to ensure that attention scores are not calculated for the padding tokens in padded sequences (e.g., 0 is often used as the padding token), but instead are calculated only for the positions in input sequences that have actual words in padded sequences; or (b) to set the attention scores for future tokens to zero, so that the model can only attend to previous tokens, as explained in the section below on decoder sub-networks).

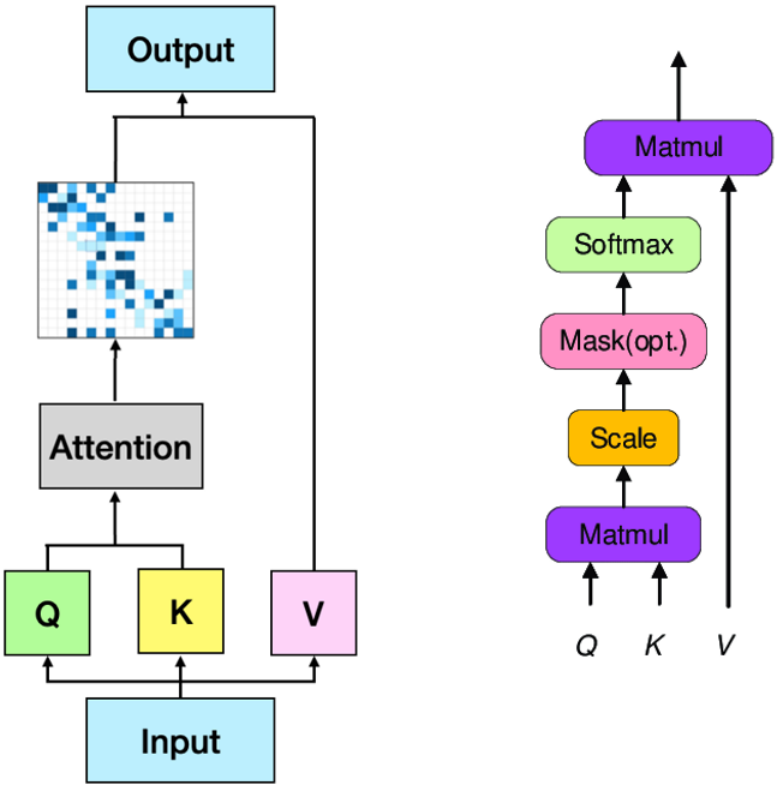


Figure: Self-attention in Transformer Networks

In conclusion, self-attention is applied to determine the meaning of the words in a sentence based on the context. That is, Transformers use the attention scores to modify the input vector representations for each word and generate a new representation based on the context of the sentence. During the training of the network, the representations of the words are updated and projected into a new embeddings space that takes the context into account.

7.20.3 20.3 Multi-Head Attention

Transformer Networks include multiple self-attention modules in their architecture. Each self-attention module is called **attention head**, and the aggregation of the outputs of multiple attention heads is called **multi-head attention**. For instance, the original Transformer model had 8 attention heads, while the GPT-3 language model has 12 attention heads.

The multi-head attention module is shown in the next figure, where the inputs are first passed through a linear layer (dense or fully-connected layer), next they are fed to the multiple attention heads, and the outputs of all attention heads are concatenated, and passed through one more linear layer.

A logical question one can ask is why are multiple attention heads needed? The reason is that multiple attention modules can learn different relationships between the words in sentences. Each module can extract context independently from the other modules, which allows to capture less obvious context and enhance the learning capabilities of the model. For example, one head may capture relationship between the nouns and numerical values in sentences, another head may focus on the relationship between the adjectives in sentences, and another head may focus on rhyming words, etc. And, if one head becomes too specialized in capturing one type of patterns, the other heads can compensate for it and provide redundancy that can improve the overall performance of the model.

Also, the computations of each attention head can be performed in parallel on different workers, which allows for accelerating the training and scaling up the models.

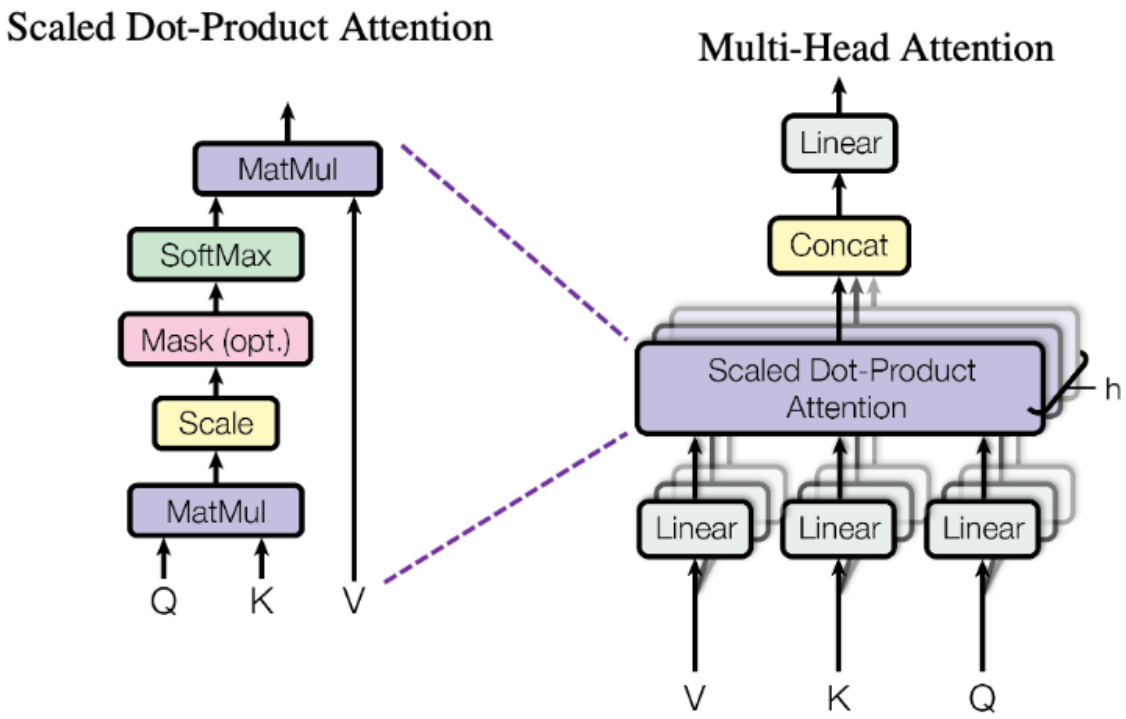


Figure: Multi-head attention

7.20.4 20.4 Encoder Block

The **Encoder Block** in Transformer Networks is shown in the next figure. It processes the input word embeddings and extracts representations in text data that can afterwards be used for different NLP tasks.

The components in the Encoder Block are:

- *Multi-head Attention layer*, which as explained, consists of multiple self-attention modules.
- *Dropout layer*, is a regular dropout layer.
- *Residual connections*, are skip connections in neural networks, where the input to a layer is added to the processed output of the layer. Residual connections were popularized in the ResNets models, as they were shown to stabilize the training and mitigate the problems of *vanishing and exploding gradients* in neural networks (i.e., they refer to cases when the gradients become too small or too large during training). In the figure, the `Add` term in the layer refers to the residual connection, which adds the input embeddings to the output of the Dropout layer.
- *Layer Normalization*, is an operation that is similar to the batch normalization in CNNs, but instead, it normalizes the outputs of each multi-head attention layer independently from the outputs of the other multi-head attention layers, and scales the data to have 0 mean and 1 standard deviation. This type of normalization is more adequate for text data. And, as we learned in the previous lectures, normalization improves the flow of gradients during training. The `Norm` term in the figure refers to the Layer Normalization operation.
- *Feed Forward network*, consists of 2 fully-connected (dense) layers that extract useful data representations.
- The Encoder Block also contains one more *Dropout layer*, and another *Add & Norm* layer that forms a residual connection for the input to the Feed Forward network and applies a layer normalization operation.

Larger Transformer networks typically include several encoder blocks in a sequence. For instance, in the original paper the authors used 6 encoder blocks.

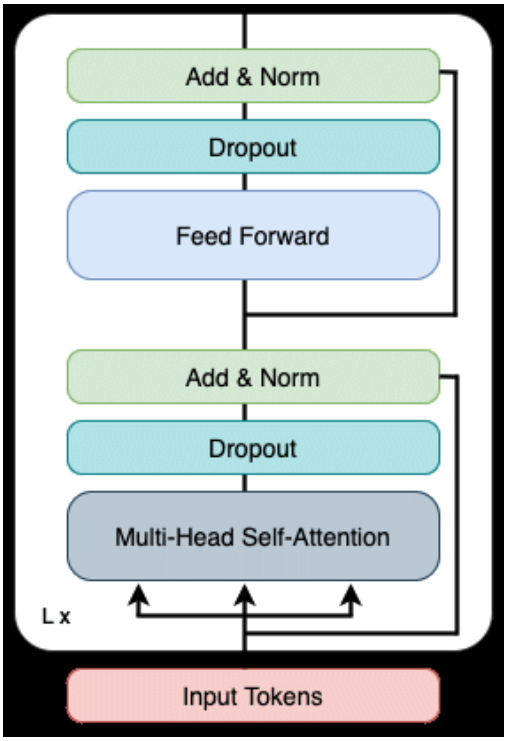


Figure: Encoder block

The implementation of the Encoder Block in Keras and TensorFlow is shown in the cell following the imported libraries.

The Encoder Block is implemented as a custom layer which is a subclass of the `Layer` class in Keras. The `__init__()` constructor method lists the definitions of the layers in the Encoder, and the method `call()` provides the forward pass with the flow of information through the layers.

- *Multi-head attention* layer is implemented in Keras, and it can be directly imported. The arguments in the layer are: `num_heads` is the number of attention heads, and `key_dim` is the dimension of the embeddings of the input tokens.
- *Dropout* and *Normalization* layers are also directly imported, with arguments `rate` for the dropout rate, and `epsilon` is a small float added to the standard deviation to avoid division by 0.
- *Feed forward network* includes 2 dense layers, with the number of neurons set to `ff_dim` and `embed_dim`, respectively.

The `call()` method specifies the forward pass of the network, and takes two parameters: `inputs` (the input embeddings to the network) and `training` (an argument which can be True or False). For the dropout layers, during the model training this argument is set to True and dropout is applied, while during inference the argument is set to False and dropout is not applied.

Each step in the `call()` method performs the data processing for one layer. Note that the `multi_head_attention` layer has as arguments the `inputs` twice, which is once for the key and once for the value in the self-attention. Also note the residual connections that are implemented in the layer normalization, e.g., the inputs are added to the output of the multi-head attention.

[ ]:
```
import tensorflow as tf
from tensorflow import keras
from keras.layers import MultiHeadAttention, LayerNormalization, Dropout, Dense,␣
↪Embedding, Layer
from keras import Sequential, Model
```

[ ]:
```
class TransformerEncoder(Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super().__init__()
        self.multi_head_attention = MultiHeadAttention(num_heads=num_heads, key_
↪dim=embed_dim)
        self.feed_forward_net = Sequential([Dense(ff_dim, activation="relu"),␣
↪Dense(embed_dim),])
        self.layer_normalization1 = LayerNormalization(epsilon=1e-6)
        self.layer_normalization2 = LayerNormalization(epsilon=1e-6)
        self.dropout1 = Dropout(rate)
        self.dropout2 = Dropout(rate)

    def call(self, inputs, training):
        multi_head_att_output = self.multi_head_attention(inputs, inputs)
        multi_head_att_dropout = self.dropout1(multi_head_att_output, training=training)
        add_norm_output_1 = self.layer_normalization1(inputs + multi_head_att_dropout)
        feed_forward_output = self.feed_forward_net(add_norm_output_1)
        feed_forward_dropout = self.dropout2(feed_forward_output, training=training)
        add_norm_output_2 = self.layer_normalization2(add_norm_output_1 + feed_forward_
↪dropout)
        return add_norm_output_2
```

7.20.5 20.5 Positional Encoding

We mentioned that Transformers use word embeddings as inputs, however, the embeddings alone don't provide information about the order of words in sentences. Understandably, the order of the words in a sentence is important, and different order of the words can convey a different meaning. To provide such information, Transformer Network introduces **positional encoding** for each word that is added to the input embedding, as shown in the next figure.

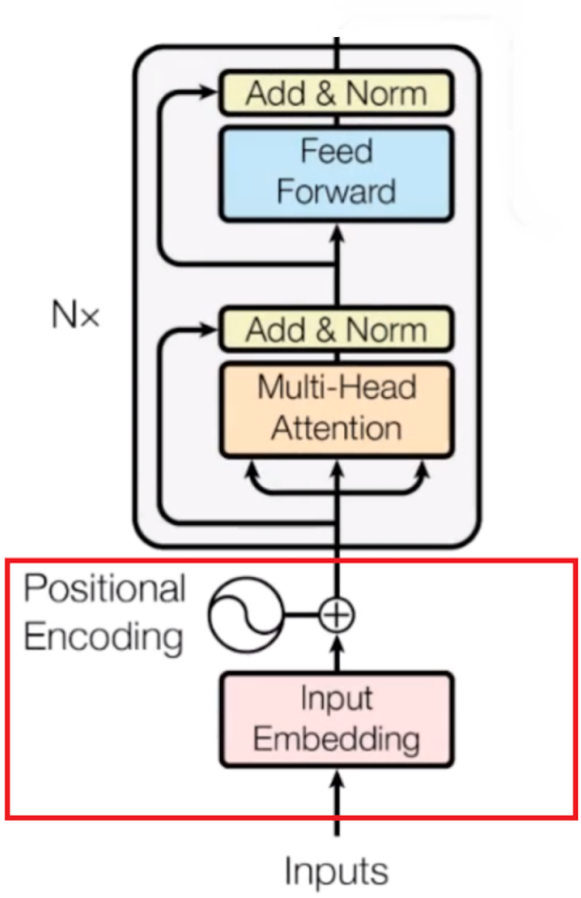


Figure: Positional encoding

There are different ways in which positional encoding can be implemented. In the original Transformer paper, the positional encoding is a vector that has the same size as the word embedding vector, and the authors used sine and cosine functions to create position vectors, which are afterwards scaled to be in the range from -1 to 1. Using such positional encoding, each encoding vector corresponds to a unique position in a sequence of words. This type is called *sinusoidal positional encoding*.

The following cell implements the addition of positional encoding to word embeddings in Keras. In this case, we will not use the approach for obtaining positional encodings based on sine and cosine functions, but instead we will use a simpler approach and learn the positional encodings in the same way the word embeddings are learned. This type of positional encoding is referred to as *learned positional encodings/embeddings*. Therefore, for both token and positional embeddings we will use the `Embedding` layer in Keras which we introduced in the previous lecture. The arguments in the `Embedding` layer are the input dimension `input_dim` and the dimension of the embedding vectors `output_dim`. For the token embeddings layer, the input dimension is the size of the vocabulary, whereas for the positional embeddings layer the input dimension is the length of the text sequences.

In the `call` method, first the length of the text sequences is assigned to `maxlen`. The function `tf.range` is similar to NumPy's `linspace` and creates numbers in the range from `start` to `limit` with a step `delta`. Next, the two separate `Embedding` layers are called, and returned is the sum of the token and positional embeddings.

[ ]:
```
class TokenAndPositionEmbedding(Layer):
    def __init__(self, maxlen, vocab_size, embed_dim):
        super().__init__()
        self.token_embeddings = Embedding(input_dim=vocab_size, output_dim=embed_dim)
        self.positional_embeddings = Embedding(input_dim=maxlen, output_dim=embed_dim)
```

(continues on next page)

(continued from previous page)

```
    def call(self, inputs):
        maxlen = tf.shape(inputs)[-1]
        positions = tf.range(start=0, limit=maxlen, delta=1)
        position_embeddings = self.positional_embeddings(positions)
        input_embeddings = self.token_embeddings(inputs)
        return input_embeddings + position_embeddings
```

7.20.6 20.6 Using a Transformer Model for Classification

Model Definition

We will now employ the layers that we defined above, to create a Transformer model for text classification.

It is a simple model that consists of the following parts:

- **Encoder**, which includes an `Input` layer that defines the maximum length of input sequences, `TokenAndPositionEmbedding` layer, and the `TransformerEncoder` layer.
- **Classifier**, which consists of a `GlobalAveragePooling1D` layer, and two `Dropout` and `Dense` layers. Global Average Pooling calculates the average value for each word, and it passes those values to the dense layers to classify the text sequences.

[ ]:
```
from keras.layers import Input, GlobalAveragePooling1D

maxlen = 200  # Maximum length of input sequences is 200 words
embed_dim = 32  # Embedding size for each token
num_heads = 2  # Number of attention heads
ff_dim = 32  # Dense layer size in the feed forward network inside transformer
vocab_size = 20000  # The size of the vocabulary is 20k words

# encoder
inputs = Input(shape=(maxlen,))
embedding_layer = TokenAndPositionEmbedding(maxlen, vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, num_heads, ff_dim)(embedding_layer)

# classifier
x = GlobalAveragePooling1D()(x)
x = Dropout(0.1)(x)
x = Dense(20, activation="relu")(x)
x = Dropout(0.1)(x)
outputs = Dense(1, activation="sigmoid")(x)

model = Model(inputs=inputs, outputs=outputs)
```

The summary of the model is shown below.

[ ]:
```
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
```

(continues on next page)

(continued from previous page)

```
 input_1 (InputLayer)        [(None, 200)]             0

 token_and_position_embeddi  (None, 200, 32)           646400
 ng (TokenAndPositionEmbedd
 ing)

 transformer_encoder (Trans  (None, 200, 32)           10656
 formerEncoder)

 global_average_pooling1d (  (None, 32)                0
 GlobalAveragePooling1D)

 dropout_2 (Dropout)         (None, 32)                0

 dense_2 (Dense)             (None, 20)                660

 dropout_3 (Dropout)         (None, 20)                0

 dense_3 (Dense)             (None, 1)                 21

=================================================================
Total params: 657737 (2.51 MB)
Trainable params: 657737 (2.51 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Loading the Dataset

Let's apply the model for sentiment analysis of the movie reviews in the IMDB database. The data is loaded from the Keras datasets, and it contains 25,000 training sequences and 25,000 validation sequences.

[ ]:
```
from keras.preprocessing.sequence import pad_sequences

(x_train, y_train), (x_val, y_val) = keras.datasets.imdb.load_data(num_words=vocab_size)
print(len(x_train), "Training sequences")
print(len(x_val), "Validation sequences")
x_train = pad_sequences(x_train, maxlen=maxlen)
x_val = pad_sequences(x_val, maxlen=maxlen)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.
↪npz
17464789/17464789 [==============================] - 1s 0us/step
25000 Training sequences
25000 Validation sequences
```

Model Training

```
[ ]: model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])

model.fit(x_train, y_train, batch_size=32, epochs=2, validation_data=(x_val, y_val))
```

```
Epoch 1/2
782/782 [==============================] - 69s 76ms/step - loss: 0.3943 - accuracy: 0.
↪8097 - val_loss: 0.2872 - val_accuracy: 0.8773
Epoch 2/2
782/782 [==============================] - 20s 26ms/step - loss: 0.1968 - accuracy: 0.
↪9239 - val_loss: 0.3153 - val_accuracy: 0.8746
```

```
<keras.src.callbacks.History at 0x7920b23f9000>
```

7.20.7 20.7 Fine-tuning a Pretrained BERT Model

BERT (Bidirectional Encoder Representations from Transformers) is a Transformer Network that can be used for variety of NLP tasks such as question answering, text classification, machine translation, etc.

In this section we will use a pretrained version of BERT and will fine-tune it for classification of news articles in the AG database (that we used in the previous lecture).

TensorFlow Hub is a repository of pretrained machine learning models, and it offers several versions of BERT such as: Small BERT, Albert, and BERT Expert. The different versions of BERT are optimized for different use cases. In our case, we will use SmallBERT.

To use this model we will need to install the TensorFlow Text library for text processing.

```
[ ]: !pip install -q tensorflow_text

import tensorflow_text as text
```

```
     6.5/6.5 MB 36.9 MB/s eta 0:00:00
```

The BERT model in TensorFlow Hub has a corresponding text preprocessing model for converting texts into tokens.

```
[ ]: import tensorflow_hub as hub
import numpy as np

bert_handle = 'https://tfhub.dev/tensorflow/small_bert/bert_en_uncased_L-4_H-512_A-8/2'
preprocessing_model = 'https://tfhub.dev/tensorflow/bert_en_uncased_preprocess/3'
```

The output of the `preprocessing model` has 3 elements:

- `input_word_ids`: token ids of the input sequences.
- `input_mask`: has value 1 for all input tokens before padding, and value 0 for the padding tokens.
- `input_type_ids`: has different values for segments in text; e.g., if there are 3 sentences in the input text, the tokens in the same sentences will have the same index.

Let's wrap `preprocessing_model` into a `hub.KerasLayer` and test it on a sample sentence.

```
[ ]: preprocess_layer = hub.KerasLayer(preprocessing_model)
```

```
[ ]: sample_news = ['Tech rumors: The tech giant Apple is working on a self driving car']

preprocessed_news = preprocess_layer(sample_news)

print('Keys:', preprocessed_news.keys())
# length of the input sequence
print('Shape:', preprocessed_news["input_word_ids"].shape)
print('Word Ids:', preprocessed_news["input_word_ids"][0,:10])
print('Input Mask:', preprocessed_news["input_mask"][0, :10])
print('Type Ids:', preprocessed_news["input_type_ids"][0, :10])
```

```
Keys: dict_keys(['input_mask', 'input_word_ids', 'input_type_ids'])
Shape: (1, 128)
Word Ids: tf.Tensor([  101  6627 11256  1024  1996  6627  5016  6207  2003  2551],␣
↪shape=(10,), dtype=int32)
Input Mask: tf.Tensor([1 1 1 1 1 1 1 1 1 1], shape=(10,), dtype=int32)
Type Ids: tf.Tensor([0 0 0 0 0 0 0 0 0 0], shape=(10,), dtype=int32)
```

Loading the Dataset

The news articles in the AG dataset are classified into 4 categories: World, Sports, Business, and Sci/Tech.

```
[ ]: import tensorflow_datasets as tfds

(train_data, val_data), info = tfds.load('ag_news_subset:1.0.0', #version 1.0.0
                                         split=('train', 'test'),
                                         with_info=True,
                                         as_supervised=True)
```

```
Downloading and preparing dataset 11.24 MiB (download: 11.24 MiB, generated: 35.79 MiB,␣
↪total: 47.03 MiB) to /root/tensorflow_datasets/ag_news_subset/1.0.0...
```

```
Dl Completed...: 0 url [00:00, ? url/s]
```

```
Dl Size...: 0 MiB [00:00, ? MiB/s]
```

```
Extraction completed...: 0 file [00:00, ? file/s]
```

```
Generating splits...:   0%|          | 0/2 [00:00<?, ? splits/s]
```

```
Generating train examples...:   0%|          | 0/120000 [00:00<?, ? examples/s]
```

```
Shuffling /root/tensorflow_datasets/ag_news_subset/1.0.0.incompleteZ0FFY4/ag_news_subset-
↪train.tfrecord*...:  ...
```

```
Generating test examples...:   0%|          | 0/7600 [00:00<?, ? examples/s]
```

```
Shuffling /root/tensorflow_datasets/ag_news_subset/1.0.0.incompleteZ0FFY4/ag_news_subset-
↪test.tfrecord*...:   ...
```

```
Dataset ag_news_subset downloaded and prepared to /root/tensorflow_datasets/ag_news_
↪subset/1.0.0. Subsequent calls will reuse this data.
```

```
[ ]: # Dataset information
class_names = info.features['label'].names
print('Classes:', class_names)
```

(continues on next page)

(continued from previous page)

```
print('Number of training samples"', info.splits['train'].num_examples)
print('Number of test samples"', info.splits['test'].num_examples)
```

```
Classes: ['World', 'Sports', 'Business', 'Sci/Tech']
Number of training samples" 120000
Number of test samples" 7600
```

[ ]:
```
buffer_size = 1000
batch_size = 32

# prepare the data
train_data = train_data.shuffle(buffer_size)
train_data = train_data.batch(batch_size).prefetch(1)
val_data = val_data.batch(batch_size).prefetch(1)
```

Model Definition with BERT

The model defined below includes an Input layer, a preprocessing layer to convert the text data into token embeddings, and a layer for the BERT model.

Afterward, the output is passed through a classifier head, which includes two dense layers and dropout layers.

[ ]:
```
# input layer
input_text = Input(shape=(), dtype=tf.string)

# preprocesing model
preprocessing_layer = hub.KerasLayer(preprocessing_model)(input_text)
# Bert model, set trainable to True
bert_encoder = hub.KerasLayer(bert_handle, trainable=True)(preprocessing_layer)

# For fine-tuning use pooled output
pooled_bert_output = bert_encoder['pooled_output']

# clasifier
x = Dense(16, activation='relu')(pooled_bert_output)
x = Dropout(0.2)(x)
final_output = Dense(4, activation='softmax')(x)

# Combine input and output
news_model = Model(input_text, final_output)
```

Model Training

Let's compile and train the model.

[ ]:
```
# compile
news_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=3e-5),
                           loss='sparse_categorical_crossentropy',
                           metrics=['accuracy'])
# train
news_model.fit(train_data, epochs=3, validation_data=val_data)
```

```
Epoch 1/3
3750/3750 [==============================] - 735s 193ms/step - loss: 0.3221 - accuracy:␣
↪0.8924 - val_loss: 0.2472 - val_accuracy: 0.9178
Epoch 2/3
3750/3750 [==============================] - 715s 191ms/step - loss: 0.2130 - accuracy:␣
↪0.9302 - val_loss: 0.2213 - val_accuracy: 0.9272
Epoch 3/3
3750/3750 [==============================] - 714s 190ms/step - loss: 0.1623 - accuracy:␣
↪0.9473 - val_loss: 0.2526 - val_accuracy: 0.9242
```

```
<keras.src.callbacks.History at 0x791f163977f0>
```

Model Evaluation

Finally, let's predict the class of two news articles.

[ ]:
```
sample_news_1 = ['Tesla, a self driving car company is also planning to make a humanoid␣
↪robot. This humanoid robot appeared dancing in the latest Tesla AI day']

predictions_1 = news_model.predict(np.array(sample_news_1))

predicted_class_1 = np.argmax(predictions_1)

print('Predicted class:', predicted_class_1)
print('Predicted class name:', class_names[predicted_class_1])
```

```
1/1 [==============================] - 0s 426ms/step
Predicted class: 3
Predicted class name: Sci/Tech
```

[ ]:
```
sample_news_2 = ["In the last weeks, there has been many transfer suprises in footbal.␣
↪Ronaldo went back to Old Trafford, "
                 "while Messi went to Paris Saint Germain to join his former colleague␣
↪Neymar."
                 "We can't wait to see these two clubs will perform in upcoming leagues"]

predictions_2 = news_model.predict(np.array(sample_news_2))

predicted_class_2 = np.argmax(predictions_2)

print('Predicted class:', predicted_class_2)
print('Predicted class name:', class_names[predicted_class_2])
```

```
1/1 [==============================] - 0s 22ms/step
Predicted class: 1
Predicted class name: Sports
```

7.20.8 20.8 Decoder Sub-network

The Transformer Network in the original paper was designed for machine translation. Differently from the text classification task where for an input text sentence the model predicts a class label, in machine translation for an input text sentence in a source language the model predicts the corresponding text sentence in a target language. Therefore, both the input and output of the model are text sequences. These type of models are called **sequence-to-sequence models**, or oftentimes this term is abbreviated to **seq2seq models**. Beside machine translation, other NLP tasks that employ seq2seq models include question answering, text summarization, dialog generation, and others.

The architecture of Transformer Networks designed to handle seq2seq tasks consists of encoder and decoder sub-networks.

- **Encoder sub-network** takes a source text sequence as an input, and extracts a useful representation of the text data.
- **Decoder sub-network** takes a target text sequence as an input, as well as it receives the intermediate representation from the encoder sub-network. The decoder combines the information from the target sequence and the encoded source sequence, and learns to predict the next word (token) in the target sequence.

This is shown in the next figure, where the French sequence "Je suis etudiant" is translated into "I am a student". The decoder outputs one word at each time step until the end-of-sequence is reached.

Figure: Decoder block

Such models that predict future values based on past observations under the assumption that the current value is dependent on previous values are called **autoregressive models**. Autoregressive text generation involves iteratively generating one token at a time, by predicting the next word or token based on the preceding words in the sequence. This approach allows the model to produce coherent and relevant responses by chatbots.

The architecture of the decoder is similar to the encoder and it is shown in the next figure. The upper part of the decoder is practically the same as the encoder, and it consists of a multi-head attention module with residual connections and layer normalization, followed by a feed-forward network with residual connections and layer normalization.

The main difference from the encoder is the *masked multi-head attention* module in the lower part of the decoder. This module is inserted before the multi-head attention module in the decoder. Masked multi-head attention module applies masking to the next words in the target sequence, so that the network does not have access to those words. That is, during training, if the model needs to predict the 4th word in a sentence, masks will be applied to all words after the 3rd word, so that the model has access only to the words 1, 2, and 3, in order to predict the 4th word. This step ensures that the model uses only the previous steps to predict the word in the next step in the target sequence. This type of mask is also referred to as **causal attention mask**, because it enforces causality by ensuring the the model only relies on information avaiable up to the current token for predicting the next token.

This also explains why in the figure below inputs to the decoder sub-network are "Outputs (shifted right)". It is because at each step, the target sequence is shifted to the right and it is fed again into the decoder. E.g., after predicting the 4th word, to predict the 5th word the input to the decoder will be words 1, 2, 3, and 4, and so on.

Finally, the output representations from the decoder are passed to a linear (dense) layer and a softmax layer, that outputs the probability for the next word in the vocabulary learned from the training dataset.

And also note the marks `Nx` in the figure. They indicate that the shown encoder and decoder blocks are repeated multiple times in the network. In the original Transformer Network, there are 6 encoder blocks, and similarly there are 6 decoder blocks. Introducing multiple blocks in the encoder and decoder sub-networks increases the learning ability as it allows the model to learn more abstract representations.

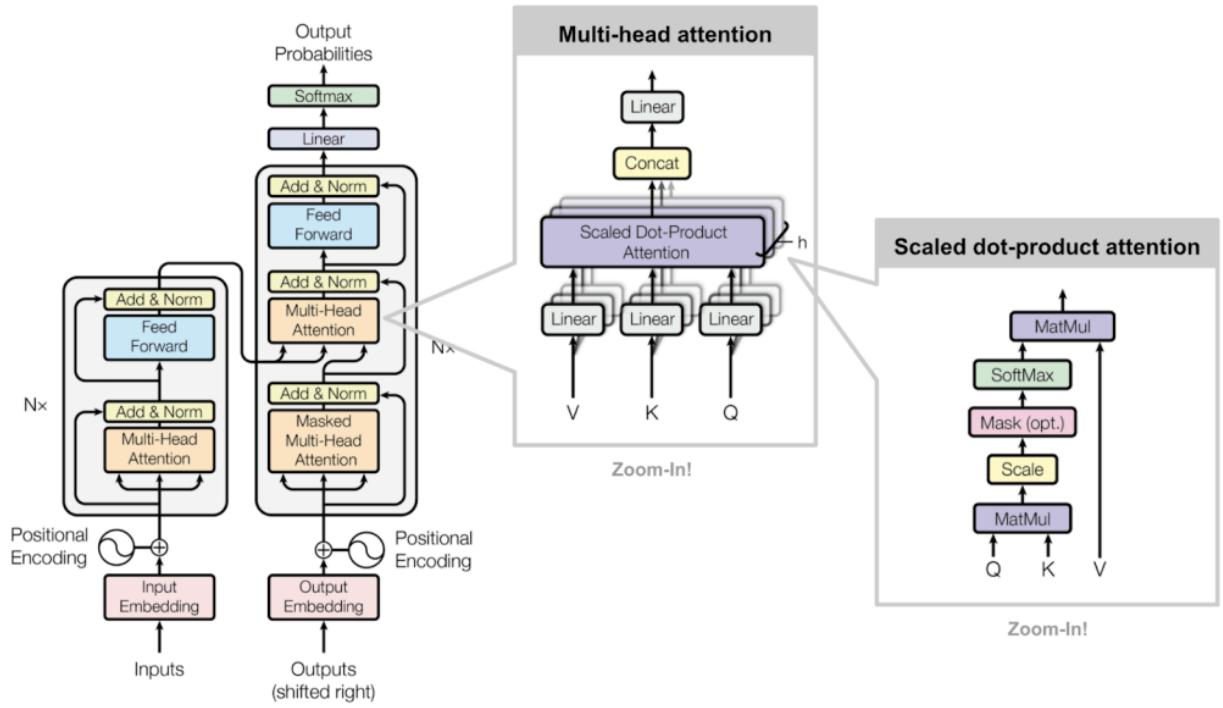


Figure: Transformer Network

Note that Recurrent Neural Networks are also a type of seq2seq models. Transformer Networks have several advantages over RNN, due to the ability to inspect entire text sequences at once, capture context in long sequences, are parallelizable, and are more powerful in general. Conversely, RNN have access only to the next token in a sequence (have difficulty finding correlations in long sequences because the information needs to pass through many processing steps), can not perform parallel computations (are slow to train), and the gradients can become unstable.

7.20.9 20.9 Vision Transformers

After the initial success of Transformer Networks in NLP, recently they have been adapted for computer vision tasks as well. The initial Transformer model for vision tasks proposed in 2021 was called **Vision Transformer (ViT)**.

The architecture of ViT is very similar to the Transformers used in NLP. However, Transformer Networks were designed for working with sequential data, while images are spatial data types. To consider each pixel in an image as a sequential token would be impractical and too time-consuming. Therefore, ViT splits images into a set of smaller image patches (16x16 pixels), and it uses the sequence of image patches as inputs to the model (i.e., each image patch is considered a token). Each image patch is first flattened to one-dimensional vector, and those vectors are afterward passed through a dense layer to learn lower-dimensional embeddings for each patch. Positional embeddings and class embeddings are added, and the sequences are fed to a standard transformer encoder. Class embeddings are vectors that correspond to different classes in the dataset. The encoder block in ViT is identical to the encoder in the original Transformer Network. The steps are depicted in the figure below.

Figure: Vision Transformer

The authors trained 3 versions of ViT, called Base (12 encoder blocks, 768 embeddings dimension, 86M parameters), Large (24 encoder blocks, 1,024 embeddings dimension, 307M parameters), and Huge (32 encoder blocks, 1,280 embeddings dimension, 632M parameters).

Various other versions of vision transformers were introduced recently, which include MaxViT (Multi-axis ViT), Swin (Shifted Window ViT), DeiT (Data-efficient image Transformer), T2T-ViT (Token-to-token ViT), and others. These

models achieved higher accuracy on many vision tasks in comparison to Convolutional Neural Networks (EffNet, ConvNeXt, NFNet). The following figure shows the accuracy on ImageNet.

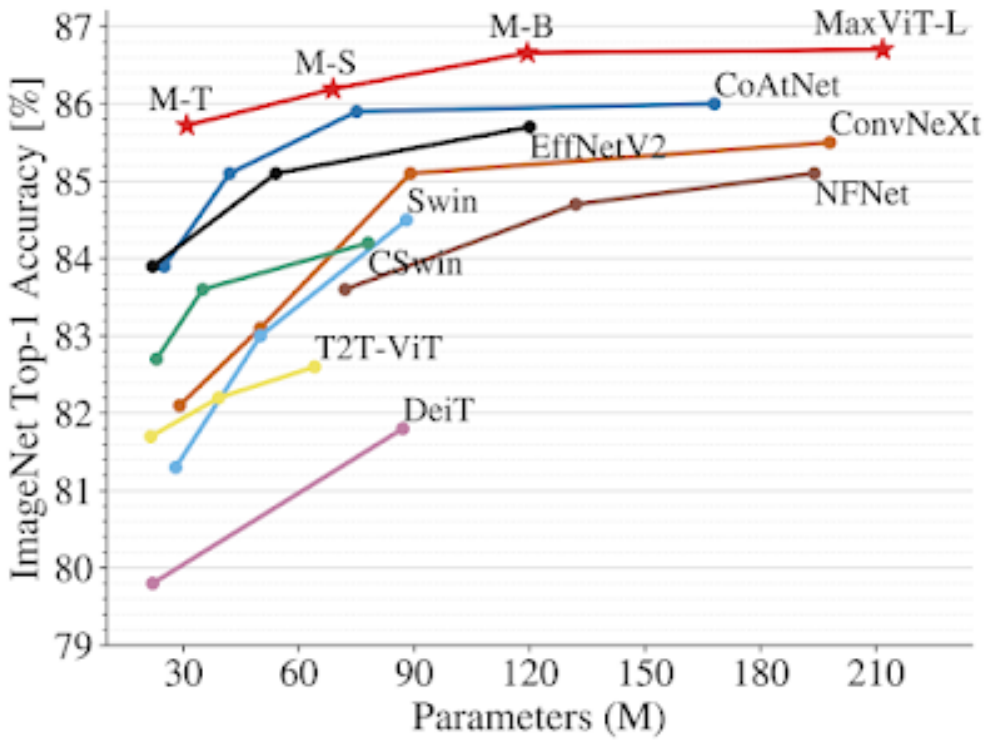


Figure: Accuracy on the ImageNet dataset

7.20.10 References

1. The Illustrated Transformer, Jay Alammar, available at: https://jalammar.github.io/illustrated-transformer/.
2. Keras Examples, Text classification with Transformer, available at: https://keras.io/examples/nlp/text_classification_with_transformer/.
3. Using Pretrained BERT for Text Classification, Jean de Dieu Nyandwi, available at: https://github.com/Nyandwi/machine_learning_complete/blob/main/9_nlp_with_tensorflow/5_using_pretrained_bert_for_text_classification.ipynb.
4. Deep Learning with Python, Francois Chollet, Second Edition, Manning Publications, 2021.
5. TensorFlow Tutorials, Neural Machine Translation with a Transformer and Keras, available at https://www.tensorflow.org/text/tutorials/transformer.
6. How the Vision Transformer (ViT) Works in 10 Minutes: An Image is Worth 16x16 Words, Nikolas Adaloglou, available at https://theaisummer.com/vision-transformer/.

BACK TO TOP

7.21 Lecture 21 - NLP with Hugging Face

7.21.1 21.1 Introduction to Hugging Face

Hugging Face (link) is a platform for Machine Learning and AI created in 2016, with the aim to "build, train, and deploy state of the art models powered by the reference open source in machine learning". Since then, Hugging Face has established itself as the main source for NLP and other Machine Learning tasks, providing open access to over 400,000 pre-trained models, datasets, and pertinent tools and resources. Hugging Face focuses on community-building around open-source machine learning tools and data. They also developed several courses on how to use their libraries for various tasks. Also note that while open access is provided to the core NLP libraries, Hugging Face also offers pricing options for access to AutoNLP libraries.

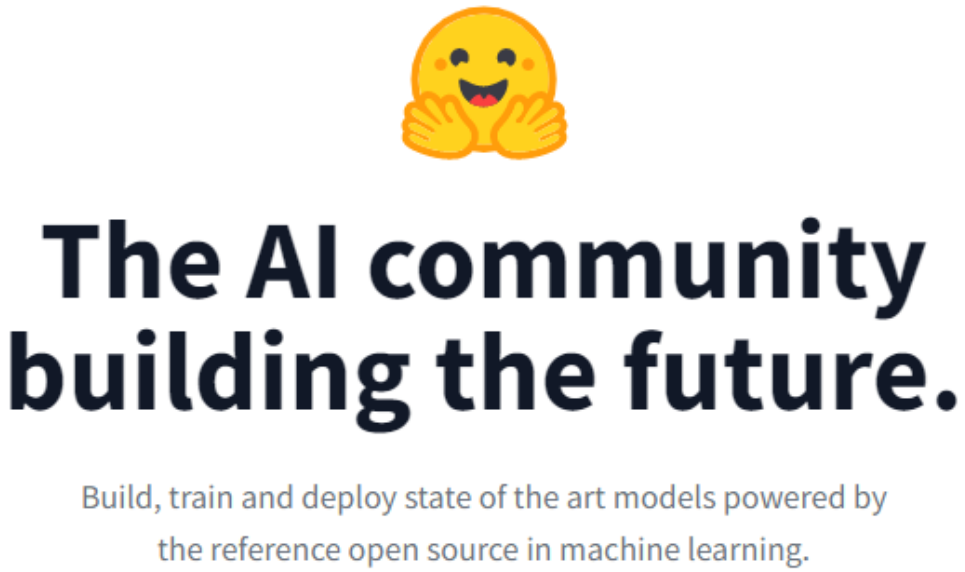


Figure: Hugging Face webpage.

Hugging Face initially focused on Transformer Networks and NLP, while recently they have expanded their libraries and tools to cover machine learning models and tasks, in general. State-of-the-art Transformer Networks are very large models, and hence, training such models from scratch is expensive and not affordable for many organizations. For example, the cost of training the GPT-4 model is estimated to over USD $100 million. Providing access to pre-trained models for transfer learning and fine-tuning to specific tasks by Hugging Face has been a significant resourse.

The core Hugging Face libraries include Transformer models, Tokenizers, Datasets, and Accelerate. Accelerate library enables distributed training with hardware acceleration devices, such as using multiple GPUs, or cloud accelerators with TPUs. In addition to these core libraries, Hugging Face provides various community resources, which include a platform for sharing models, code versioning, Spaces allow sharing apps developed with Hugging Face libraries and browsing apps created by others, etc.

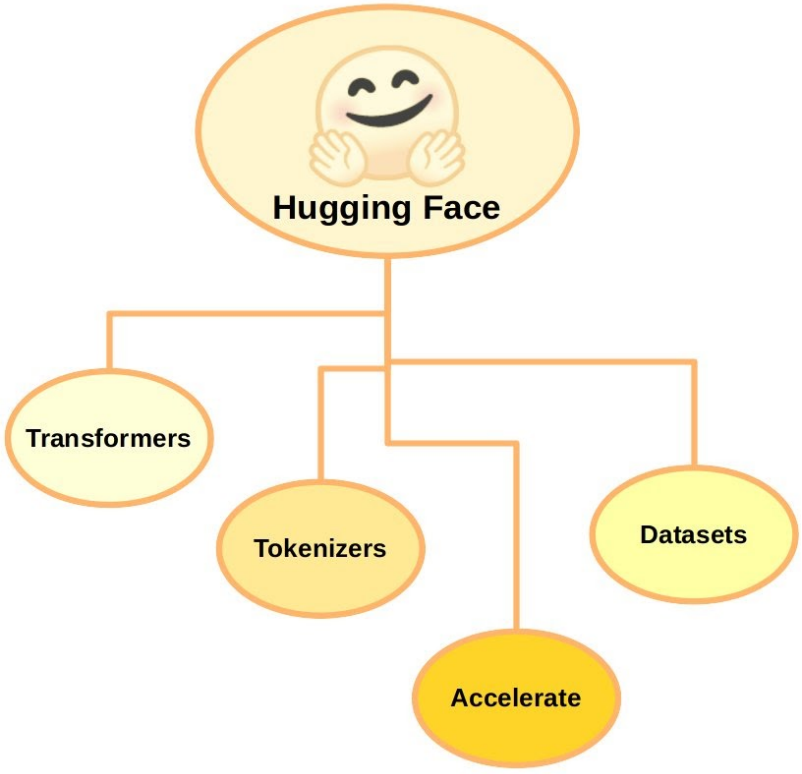


Figure: Hugging Face libraries.

The key characteristics of these libraries include:

- Ease of use and simplicity, where downloading and using state-of-the-art NLP models can be done with a few lines of code.
- Flexibility, since all models are implemented either using the `nn.Module` in PyTorch or `tf.keras.Model` in TensorFlow, allowing for easy model integration with these popular frameworks.

7.21.2 21.2 Hugging Face Pipelines

Hugging Face uses **Pipelines** as an API that through the `pipeline()` method allows performing inference over a variety of tasks.

The `pipeline()` method has the following syntax:

```
from transformers import pipeline

# Pipeline to use a default model & tokenizer for a given task
pipeline("<task-name>")

# Pipeline to use an existing or custom model
pipeline("<task-name>", model="<model_name>")

# Pipeline to use an existing or custom model and tokenizer
pipeline('<task-name>', model='<model name>', tokenizer='<tokenizer_name>')
```

Among the currently available task pipelines are:

- Sentiment-analysis
- Question-answering
- Translation
- Summarization
- Text-generation
- NER (named entity recognition)
- Zero-shot-classification
- Fill-mask

7.21.3 21.3 Pipelines for NLP Tasks

In this section, we will examine examples on how to use the `pipeline("<task-name>")` method with different NLP tasks. As we mentioned, if we don't provide the names for the used model and tokenizer, the pipeline will assign a default language model and tokenizer to complete the task, and it will download the model parameters and other required elements to perform text generation.

To use the Transformer library by Hugging Face we will need to first install it, since it is not preinstalled in Google Colab.

[ ]:
```
!pip install -q transformers datasets
```

```
     7.9/7.9 MB 51.6 MB/s eta 0:00:00
     493.7/493.7 kB 43.0 MB/s eta 0:00:00
     302.0/302.0 kB 36.1 MB/s eta 0:00:00
     3.8/3.8 MB 105.6 MB/s eta 0:00:00
     1.3/1.3 MB 53.2 MB/s eta 0:00:00
     115.3/115.3 kB 12.5 MB/s eta 0:00:00
     134.8/134.8 kB 13.2 MB/s eta 0:00:00
     295.0/295.0 kB 30.1 MB/s eta 0:00:00
```

21.3.1 Sentiment Analysis

The first example uses `pipeline()` for sentiment analysis. When the cell is executed, the pipeline will select a default pretrained model for sentiment analysis in English, it will download the model and the related tokenizer, and it will instantiate a text classifier object.

We saw examples of sentiment analysis in the previous lectures, where the goal was to classify the sentiment in movie reviews text as positive or negative.

[ ]:
```
from transformers import pipeline

classifier = pipeline("sentiment-analysis")
```

```
No model was supplied, defaulted to distilbert-base-uncased-finetuned-sst-2-english and␣
↪revision af0f99b (https://huggingface.co/distilbert-base-uncased-finetuned-sst-2-
↪english).
Using a pipeline without specifying a model name and revision in production is not␣
↪recommended.
```

```
Downloading (...)lve/main/config.json:   0%|          | 0.00/629 [00:00<?, ?B/s]
Downloading model.safetensors:   0%|          | 0.00/268M [00:00<?, ?B/s]
Downloading (...)okenizer_config.json:   0%|          | 0.00/48.0 [00:00<?, ?B/s]
Downloading (...)solve/main/vocab.txt:   0%|          | 0.00/232k [00:00<?, ?B/s]
```

In the next cell, the classifier is applied to a sentence. The output is the predicted label and the confidence score.

```
[ ]: classifier("I fully understand what you are saying.")
```

```
[{'label': 'POSITIVE', 'score': 0.9996806383132935}]
```

The pipeline allows to pass multiple sentences, and it will return a sentiment label and confidence score for each sentence.

```
[ ]: classifier(
    ["I've been waiting for a HuggingFace course my whole life.", "I hate this so much!"]
)
```

```
[{'label': 'POSITIVE', 'score': 0.9598048329353333},
 {'label': 'NEGATIVE', 'score': 0.9994558691978455}]
```

21.3.2 Question Answering

This pipeline answers questions using information from a given context. Such pipeline can be very useful when we are dealing with long text data and finding answers to questions in the document can take time.

```
[ ]: question_answerer = pipeline("question-answering")
```

```
No model was supplied, defaulted to distilbert-base-cased-distilled-squad and revision␣
↪626af31 (https://huggingface.co/distilbert-base-cased-distilled-squad).
Using a pipeline without specifying a model name and revision in production is not␣
↪recommended.
Downloading (...)lve/main/config.json:   0%|          | 0.00/473 [00:00<?, ?B/s]
Downloading model.safetensors:   0%|          | 0.00/261M [00:00<?, ?B/s]
Downloading (...)okenizer_config.json:   0%|          | 0.00/29.0 [00:00<?, ?B/s]
Downloading (...)solve/main/vocab.txt:   0%|          | 0.00/213k [00:00<?, ?B/s]
Downloading (...)/main/tokenizer.json:   0%|          | 0.00/436k [00:00<?, ?B/s]
```

We can provide inputs to the pipeline as a dictionary with `question` and `context` as keys. The model extracts information from the provided context and returns a dictionary with a confidence score, start and end characters of the answer in the context, and the answer. Also note that the model does not generate new text to answer the questions, but instead it searches for the answer in the supplied context sequence.

```
[ ]: input_1 = {
    "question" : "What didn't cross the street?",
    "context" : "The animal didn't cross the street because it was too tired",
    }

question_answerer(input_1)
```

```
{'score': 0.7553669214248657, 'start': 0, 'end': 10, 'answer': 'The animal'}
```

[ ]:
```
input_2 = {
    "question" : "Why the animal didn't cross the street?",
    "context" : "The animal didn't cross the street because it was too wide",
    }

question_answerer(input_2)
```

```
{'score': 0.607613742351532,
 'start': 43,
 'end': 58,
 'answer': 'it was too wide'}
```

21.3.3 Machine Translation

For machine translation, we can provide source and target languages in the pipeline, as in the next cell where the task `"translation_en_to_fr"` is to translate text from English to French. Although this pipeline can work with several languages, most often, machine translation requires to specify the name of the used language model, and only for several special cases it can work by specifying only the task name.

[ ]:
```
translator = pipeline("translation_en_to_fr")
```

```
No model was supplied, defaulted to t5-base and revision 686f1db (https://huggingface.co/
↪t5-base).
Using a pipeline without specifying a model name and revision in production is not␣
↪recommended.
```

```
Downloading (...)lve/main/config.json:   0%|          | 0.00/1.21k [00:00<?, ?B/s]
```

```
Downloading model.safetensors:   0%|          | 0.00/892M [00:00<?, ?B/s]
```

```
Downloading (...)neration_config.json:   0%|          | 0.00/147 [00:00<?, ?B/s]
```

```
Downloading (...)ve/main/spiece.model:   0%|          | 0.00/792k [00:00<?, ?B/s]
```

```
Downloading (...)/main/tokenizer.json:   0%|          | 0.00/1.39M [00:00<?, ?B/s]
```

```
/usr/local/lib/python3.10/dist-packages/transformers/models/t5/tokenization_t5_fast.py:
↪160: FutureWarning: This tokenizer was incorrectly instantiated with a model max␣
↪length of 512 which will be corrected in Transformers v5.
For now, this behavior is kept to avoid breaking backwards compatibility when padding/
↪encoding with `truncation is True`.
- Be aware that you SHOULD NOT rely on t5-base automatically truncating your input to␣
↪512 when padding/encoding.
- If you want to encode/pad to sequences longer than 512 you can either instantiate this␣
↪tokenizer with `model_max_length` or pass `max_length` when encoding/padding.
- To avoid this warning, please instantiate this tokenizer with `model_max_length` set␣
↪to your preferred value.
  warnings.warn(
```

[ ]:
```
translator("I am a student")
```

```
[{'translation_text': 'Je suis un étudiant'}]
```

[ ]:
```
translator("Peyton Manning became the first quarterback ever to lead two different teams␣
↪to multiple Super Bowls.")
```

```
[{'translation_text': 'Peyton Manning est devenu le premier quarterback à conduire deux␣
↪équipes différentes à plusieurs Super Bowls.'}]
```

21.3.4 Text Summarization

Text summarization reduces a longer text into a shorter summary.

[ ]:
```
summarizer = pipeline("summarization")
```

```
No model was supplied, defaulted to sshleifer/distilbart-cnn-12-6 and revision a4f8f3e␣
↪(https://huggingface.co/sshleifer/distilbart-cnn-12-6).
Using a pipeline without specifying a model name and revision in production is not␣
↪recommended.
```

```
Downloading (...)lve/main/config.json:   0%|          | 0.00/1.80k [00:00<?, ?B/s]
```

```
Downloading pytorch_model.bin:   0%|          | 0.00/1.22G [00:00<?, ?B/s]
```

```
Downloading (...)okenizer_config.json:   0%|          | 0.00/26.0 [00:00<?, ?B/s]
```

```
Downloading (...)olve/main/vocab.json:   0%|          | 0.00/899k [00:00<?, ?B/s]
```

```
Downloading (...)olve/main/merges.txt:   0%|          | 0.00/456k [00:00<?, ?B/s]
```

[ ]:
```
summarizer(
    """
    America has changed dramatically during recent years. Not only has the number of
    graduates in traditional engineering disciplines such as mechanical, civil,
    electrical, chemical, and aeronautical engineering declined, but in most of
    the premier American universities engineering curricula now concentrate on
    and encourage largely the study of engineering science. As a result, there
    are declining offerings in engineering subjects dealing with infrastructure,
    the environment, and related issues, and greater concentration on high
    technology subjects, largely supporting increasingly complex scientific
    developments. While the latter is important, it should not be at the expense
    of more traditional engineering.

    Rapidly developing economies such as China and India, as well as other
    industrial countries in Europe and Asia, continue to encourage and advance
    the teaching of engineering. Both China and India, respectively, graduate
    six and eight times as many traditional engineers as does the United States.
    Other industrial countries at minimum maintain their output, while America
    suffers an increasingly serious decline in the number of engineering graduates
    and a lack of well-educated engineers.
"""
)
```

```
[{'summary_text': ' The number of engineering graduates in the United States has␣
↪declined in recent years . China and India graduate six and eight times as many␣
↪traditional engineers as the U.S. does . Rapidly developing economies such as China␣
↪continue to encourage and advance the teaching of engineering . There are declining␣
↪offerings in engineering subjects dealing with infrastructure, infrastructure, the␣
↪environment, and related issues .'}]
```

Specifying the `min_length` and `max_length` arguments allows to control the length of the summary.

[ ]:
```
summarizer(
    """
    Flooding on the Yangtze river remains serious although water levels on parts of the␣
↪river decreased
    today, according to the state headquarters of flood control and drought relief .
    """, min_length=8, max_length=20)
```

```
[{'summary_text': ' Flooding on the Yangtze river remains serious although water levels␣
↪on parts of the'}]
```

[ ]:
```
summarizer(
    """BAGHDAD -- Archaeologists in northern Iraq last week unearthed 2,700-year-old␣
↪rock carvings featuring war scenes and trees from the Assyrian Empire, an␣
↪archaeologist said Wednesday.

    The carvings on marble slabs were discovered by a team of experts in Mosul, Iraq's␣
↪second-largest city, who have been working to restore the site of the ancient Mashki␣
↪Gate, which was bulldozed by Islamic State group militants in 2016.

    Fadhil Mohammed, head of the restoration works, said the team was surprised by␣
↪discovering "eight murals with inscriptions, decorative drawings and writings."

    Mashki Gate was one of the largest gates of Nineveh, an ancient Assyrian city of␣
↪this part of the historic region of Mesopotamia.

    The discovered carvings show, among other things, a fighter preparing to fire an␣
↪arrow while others show palm trees.
    """)
```

```
[{'summary_text': ' The carvings on marble slabs were discovered by a team of experts in␣
↪Mosul, Iraq's second-largest city . They have been working to restore the site of the␣
↪ancient Mashki Gate, which was bulldozed by Islamic State group militants in 2016 .␣
↪Mashki gate was one of the largest gates of Nineveh, an ancient Assyrian city of this␣
↪part of Mesopotamia .'}]
```

21.3.5 Text Generation

In this example, we will use the "`text-generation`" pipeline to generate text based on a provided prompt.

[ ]:
```
generator = pipeline("text-generation")
```

```
No model was supplied, defaulted to gpt2 and revision 6c0e608 (https://huggingface.co/
↪gpt2).
Using a pipeline without specifying a model name and revision in production is not␣
↪recommended.
```

```
Downloading (...)lve/main/config.json:   0%|          | 0.00/665 [00:00<?, ?B/s]
```

```
Downloading model.safetensors:   0%|          | 0.00/548M [00:00<?, ?B/s]
```

```
Downloading (...)neration_config.json:   0%|          | 0.00/124 [00:00<?, ?B/s]
```

```
Downloading (...)olve/main/vocab.json:   0%|          | 0.00/1.04M [00:00<?, ?B/s]
```

```
Downloading (...)olve/main/merges.txt:   0%|          | 0.00/456k [00:00<?, ?B/s]
Downloading (...)/main/tokenizer.json:   0%|          | 0.00/1.36M [00:00<?, ?B/s]
```

Now let's provide a prompt text to the generator object, and the generator will continue the text. Note that text generation involves randomness, so some of the outputs will not be perfect. And admittedly, this is one of the most difficult NLP tasks.

[ ]:
```
outputs_1 = generator("In this course, we will teach you how to")

print(outputs_1[0]['generated_text'])
```

```
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
```

```
In this course, we will teach you how to design a powerful software architecture to␣
↪handle high complexity Web application developers. We will show you how to use a␣
↪simple interface to implement the dataflow concepts and we will be showing how simple␣
↪these concepts can be
```

[ ]:
```
outputs_2 = generator("Niagara Falls is a city located in")

print(outputs_2[0]['generated_text'])
```

```
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
```

```
Niagara Falls is a city located in the northeast corner of New York town. It is located␣
↪within walking distance to the east of New York's main highway, and adjacent to␣
↪Interstate 90 on the Queens Bridge. In the 1920s, when Niagara Falls
```

[ ]:
```
outputs_3 = generator("Niagara Falls is a famous world attractation")

print(outputs_3[0]['generated_text'])
```

```
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
```

```
Niagara Falls is a famous world attractation so you can see the amazing waterfalls,␣
↪creeks, or valleys below. It's also popular because it offers great views of the world
↪'s most famous waterfall.

In the summer, you
```

21.3.6 Named Entity Recognition

Named Entity Recognition (NER), also known as named entity tagging, is a task of identifying parts of the input that represent entities. Examples of entities are:

- Location (LOC)
- Organizations (ORG)
- Persons (PER)
- Miscellaneous entities (MISC)

[ ]:
```
ner = pipeline("ner", grouped_entities=True)
```

```
No model was supplied, defaulted to dbmdz/bert-large-cased-finetuned-conll03-english and␣
↪revision f2482bf (https://huggingface.co/dbmdz/bert-large-cased-finetuned-conll03-
↪english).
Using a pipeline without specifying a model name and revision in production is not␣
↪recommended.
```

```
Downloading (...)lve/main/config.json:   0%|          | 0.00/998 [00:00<?, ?B/s]
```

```
Downloading model.safetensors:   0%|          | 0.00/1.33G [00:00<?, ?B/s]
```

```
Some weights of the model checkpoint at dbmdz/bert-large-cased-finetuned-conll03-english␣
↪were not used when initializing BertForTokenClassification: ['bert.pooler.dense.bias',
↪'bert.pooler.dense.weight']
- This IS expected if you are initializing BertForTokenClassification from the␣
↪checkpoint of a model trained on another task or with another architecture (e.g.␣
↪initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertForTokenClassification from the␣
↪checkpoint of a model that you expect to be exactly identical (initializing a␣
↪BertForSequenceClassification model from a BertForSequenceClassification model).
```

```
Downloading (...)okenizer_config.json:   0%|          | 0.00/60.0 [00:00<?, ?B/s]
```

```
Downloading (...)solve/main/vocab.txt:   0%|          | 0.00/213k [00:00<?, ?B/s]
```

```
/usr/local/lib/python3.10/dist-packages/transformers/pipelines/token_classification.py:
↪169: UserWarning: `grouped_entities` is deprecated and will be removed in version v5.0.
↪0, defaulted to `aggregation_strategy="simple"` instead.
  warnings.warn(
```

[ ]:
```
text_1 = "Abraham Lincoln was a president who lived in the United States."

print(ner(text_1))
```

```
[{'entity_group': 'PER', 'score': 0.9988935, 'word': 'Abraham Lincoln', 'start': 0, 'end
↪': 15}, {'entity_group': 'LOC', 'score': 0.99965084, 'word': 'United States', 'start':␣
↪49, 'end': 62}]
```

Or, we can use Pandas to display the output.

[ ]:
```
import pandas as pd

pd.DataFrame(ner(text_1))
```

	entity_group	score	word	start	end
0	PER	0.998893	Abraham Lincoln	0	15
1	LOC	0.999651	United States	49	62

[ ]:
```
text_2 = """BAGHDAD -- Archaeologists in northern Iraq last week unearthed 2,700-year-
↪old rock carvings featuring war scenes and trees from the Assyrian Empire, an␣
↪archaeologist said Wednesday.

    The carvings on marble slabs were discovered by a team of experts in Mosul, Iraq's␣
↪second-largest city, who have been working to restore the site of the ancient Mashki␣
↪Gate, which was bulldozed by Islamic State group militants in 2016.

    Fadhil Mohammed, head of the restoration works, said the team was surprised by␣
↪discovering "eight murals with inscriptions, decorative drawings and writings."
```

(continues on next page)

(continued from previous page)

```
    Mashki Gate was one of the largest gates of Nineveh, an ancient Assyrian city of␣
↪this part of the historic region of Mesopotamia.

    The discovered carvings show, among other things, a fighter preparing to fire an␣
↪arrow while others show palm trees.
    """

pd.DataFrame(ner(text_2))
```

	entity_group	score	word	start	end
0	LOC	0.434805	BA	0	2
1	LOC	0.999473	Iraq	38	42
2	MISC	0.893631	Assyrian	132	140
3	LOC	0.782092	Empire	141	147
4	LOC	0.999238	Mosul	256	261
5	LOC	0.999156	Iraq	263	267
6	LOC	0.971527	Mashki Gate	348	359
7	ORG	0.997262	Islamic State	384	397
8	PER	0.999300	Fadhil Mohammed	428	443
9	LOC	0.974939	Mashki Gate	592	603
10	LOC	0.975865	Nineveh	636	643
11	MISC	0.994617	Assyrian	656	664
12	LOC	0.976547	Mesopotamia	709	720

21.3.7 Zero-shot Classification

Zero-shot classification is a task to classify text documents, where the term *zero-shot* classification refers to tasks for which a language model has not been trained. I.e., the model was not trained to classify documents using the provided type of labels in the next example.

[ ]:
```
classifier = pipeline("zero-shot-classification")
```

```
No model was supplied, defaulted to facebook/bart-large-mnli and revision c626438 (https:
↪//huggingface.co/facebook/bart-large-mnli).
Using a pipeline without specifying a model name and revision in production is not␣
↪recommended.
```

```
Downloading (...)lve/main/config.json:   0%|          | 0.00/1.15k [00:00<?, ?B/s]
Downloading model.safetensors:   0%|          | 0.00/1.63G [00:00<?, ?B/s]
Downloading (...)okenizer_config.json:   0%|          | 0.00/26.0 [00:00<?, ?B/s]
Downloading (...)olve/main/vocab.json:   0%|          | 0.00/899k [00:00<?, ?B/s]
Downloading (...)olve/main/merges.txt:   0%|          | 0.00/456k [00:00<?, ?B/s]
Downloading (...)/main/tokenizer.json:   0%|          | 0.00/1.36M [00:00<?, ?B/s]
```

The pipeline allows us to list `candidate labels` to be used for the classification. For this example, the model returned confidence scores for each category, and the highest probability was assigned to the "sports" category.

[ ]:
```
classifier(
    "Peyton Manning became the first quarterback ever to lead two different teams to␣
↪multiple Super Bowls.",
```

(continues on next page)

(continued from previous page)

```
    candidate_labels=["education", "politics", "business", "sports"],
)
```

```
{'sequence': 'Peyton Manning became the first quarterback ever to lead two different␣
↪teams to multiple Super Bowls.',
 'labels': ['sports', 'business', 'education', 'politics'],
 'scores': [0.9866245985031128,
  0.006729820277541876,
  0.0034621653612703085,
  0.003183382796123624]}
```

21.3.8 Mask Filling

The pipeline with the `fill-mask` task is used to fill in blanks in an input text.

[ ]:
```
mask_filling = pipeline("fill-mask")
```

```
No model was supplied, defaulted to distilroberta-base and revision ec58a5b (https://
↪huggingface.co/distilroberta-base).
Using a pipeline without specifying a model name and revision in production is not␣
↪recommended.
```

```
Downloading (...)lve/main/config.json:   0%|          | 0.00/480 [00:00<?, ?B/s]
```

```
Downloading model.safetensors:   0%|          | 0.00/331M [00:00<?, ?B/s]
```

```
Some weights of the model checkpoint at distilroberta-base were not used when␣
↪initializing RobertaForMaskedLM: ['roberta.pooler.dense.bias', 'roberta.pooler.dense.
↪weight']
- This IS expected if you are initializing RobertaForMaskedLM from the checkpoint of a␣
↪model trained on another task or with another architecture (e.g. initializing a␣
↪BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing RobertaForMaskedLM from the checkpoint of␣
↪a model that you expect to be exactly identical (initializing a␣
↪BertForSequenceClassification model from a BertForSequenceClassification model).
```

```
Downloading (...)olve/main/vocab.json:   0%|          | 0.00/899k [00:00<?, ?B/s]
```

```
Downloading (...)olve/main/merges.txt:   0%|          | 0.00/456k [00:00<?, ?B/s]
```

```
Downloading (...)/main/tokenizer.json:   0%|          | 0.00/1.36M [00:00<?, ?B/s]
```

We can provide the `top_k` argument to indicate the number of returned answers.

[ ]:
```
mask_filling("Abraham Lincoln was a <mask> who lived in the United States.", top_k=5)
```

```
[{'score': 0.3327265977859497,
  'token': 3661,
  'token_str': ' Democrat',
  'sequence': 'Abraham Lincoln was a Democrat who lived in the United States.'},
 {'score': 0.18091031908988953,
  'token': 1172,
  'token_str': ' Republican',
  'sequence': 'Abraham Lincoln was a Republican who lived in the United States.'},
 {'score': 0.033906374126672745,
```

(continues on next page)

(continued from previous page)

```
  'token': 16495,
  'token_str': ' Jew',
  'sequence': 'Abraham Lincoln was a Jew who lived in the United States.'},
 {'score': 0.028415339067578316,
  'token': 24156,
  'token_str': ' Presbyterian',
  'sequence': 'Abraham Lincoln was a Presbyterian who lived in the United States.'},
 {'score': 0.02462911792099476,
  'token': 11593,
  'token_str': ' physician',
  'sequence': 'Abraham Lincoln was a physician who lived in the United States.'}]
```

[ ]:
```
mask_filling("Flooding on the Yangtze river remains serious although <mask> levels on␣
↪parts of the river decreased today.", top_k=2)
```

```
[{'score': 0.2489088624715805,
  'token': 514,
  'token_str': ' water',
  'sequence': 'Flooding on the Yangtze river remains serious although water levels on␣
↪parts of the river decreased today.'},
 {'score': 0.12597304582595825,
  'token': 11747,
  'token_str': ' oxygen',
  'sequence': 'Flooding on the Yangtze river remains serious although oxygen levels on␣
↪parts of the river decreased today.'}]
```

7.21.4 21.4 Tokenizers

Tokenizers library in Hugging Face is used to split input text data into tokens (e.g., words, characters, N-grams), and map the tokens to integers. When we use a pretrained model from Hugging Face for a downstream task, our text data needs to be preprocessed in the same way as the training data used with the model. Therefore, we will need to download the tokenizer for that specific model.

Let's consider the model `"distilbert-base-uncased"`, which is a version of the BERT transformer model, which takes case-insensitive English text as input data. Next, we will download the tokenizer for this model by using the `AutoTokenizer` class and its method `from_pretrained()`. By using `AutoTokenizer` we don't need to manually download and manage the tokenizer files.

Or, we can also specify the tokenizer name for the model that we wish to use. For instance, for the BERT model, Hugging Face has a `BertTokenizer` that can be directly imported from the `transformers` package.

[ ]:
```
from transformers import AutoTokenizer

model = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model)
```

```
Downloading (...)okenizer_config.json:   0%|          | 0.00/28.0 [00:00<?, ?B/s]
Downloading (...)lve/main/config.json:   0%|          | 0.00/483 [00:00<?, ?B/s]
Downloading (...)solve/main/vocab.txt:   0%|          | 0.00/232k [00:00<?, ?B/s]
Downloading (...)/main/tokenizer.json:   0%|          | 0.00/466k [00:00<?, ?B/s]
```

Let's now use the `tokenizer` to convert sentences into a sequence of integers, and display the output.

The ouput of the tokenizer is a dictionary consisting of two key-value pairs:

- `input_ids`, a list of integers, where each index identifies a token. The indexing is based on the vocabulary of the training data that was used to train the model `"distilbert-base-uncased"`.
- `attention_mask`, a list of 1's or 0's, to indicate padding of the text sequence. This sentence does not have padding, since all elements have an attention mask of 1's. The attention mask ensures that the attention mechanism in Transformer is applied only to the real tokens, and the padding tokens are ignored.

[ ]:
```
output_tokens_1 = tokenizer('Tokenizing text is easy.')
print(output_tokens_1)
```

```
{'input_ids': [101, 19204, 6026, 3793, 2003, 3733, 1012, 102], 'attention_mask': [1, 1,␣
↪1, 1, 1, 1, 1, 1]}
```

Notethat the above output has 8 tokens, although the input sentence has 4 words and the period mark. To understand better how the tokenization was performed, in the next cell we used the method `covert_ids_to_tokens()` to obtain the text for each integer value. Now we can see that the tokenizer places special tokens at the beginning and end of each sequence. `[CLS]` is placed at the beginning (it stands for Classification), and `[SEP]` is placed at the end of the sequence (it stands for Separate).

Also note that the gerund verb "tokenizing" is split into `'token'` and `'##izing'`. Using two tokens for the word allows to work with smaller vocabularies. I.e., instead of considering `token` and `tokenization` as two different words, by splitting the word into the root `token` and the suffix `ization`, the model will use two tokens that have a distinct semantic meaning. This approach of decomposing long words into subwords is especially efficient with some languages where one can form very long words by chaining simple subwords.

[ ]:
```
tokenizer.convert_ids_to_tokens(output_tokens_1.input_ids)
```

```
['[CLS]', 'token', '##izing', 'text', 'is', 'easy', '.', '[SEP]']
```

Most tokenizers in Hugging Face assign integers for the special tokens between 100 and 103.

These special tokens include:

- [PAD], padding.
- [UNK], unknown token.
- [CLS], sequence beginning.
- [SEP], sequence end.
- [MASK], masked tokens.

[ ]:
```
tokenizer.convert_ids_to_tokens([0, 100, 101, 102, 103])
```

```
['[PAD]', '[UNK]', '[CLS]', '[SEP]', '[MASK]']
```

Another simple example is provided next. We can see that the word 'Transformer' is tokenized as 'transform' + '##er'.

[ ]:
```
output_tokens_2 = tokenizer('Using a Transformer network in Hugging Face is simple')
print(output_tokens_2)
```

```
{'input_ids': [101, 2478, 1037, 10938, 2121, 2897, 1999, 17662, 2227, 2003, 3722, 102],
↪'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

[ ]:
```
tokenizer.convert_ids_to_tokens(output_tokens_2.input_ids)
```

```
['[CLS]',
 'using',
 'a',
 'transform',
 '##er',
 'network',
 'in',
 'hugging',
 'face',
 'is',
 'simple',
 '[SEP]']
```

Tokenizers can be applied to multiple text sequences. The argument `padding=True` is used below to pad the sequences to the longest sequence. Note below that 0's are added to pad the second sentence.

[ ]:
```
text_3 =  ["I've been waiting for a HuggingFace course my whole life.", "I hate this so much!"]

output_tokens_3 = tokenizer(text_3, padding=True)
```

[ ]:
```
print(output_tokens_3)
```

```
{'input_ids': [[101, 1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662, 12172, 2607, 2026, 2878, 2166, 1012, 102], [101, 1045, 5223, 2023, 2061, 2172, 999, 102, 0, 0, 0, 0, 0, 0, 0, 0]], 'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]]}
```

The output is more readable if we print it line by line.

[ ]:
```
print("Input IDs")
for item in output_tokens_3.input_ids:
    print(item)

print("Attention Mask")
for item in output_tokens_3.attention_mask:
    print(item)
```

```
Input IDs
[101, 1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662, 12172, 2607, 2026, 2878, 2166, 1012, 102]
[101, 1045, 5223, 2023, 2061, 2172, 999, 102, 0, 0, 0, 0, 0, 0, 0, 0]
Attention Mask
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
```

Note also that each sequence of tokens begins with 101 ('[CLS]') and ends with 102 ('[SEP]').

If `max_length` is provided, the tokenizer will truncate longer sentences to the specified length, as in the example in the next cell.

[ ]:
```
output_tokens_4 = tokenizer(text_3, padding=False, max_length=10)

print("Input IDs")
```

(continues on next page)

(continued from previous page)

```
for item in output_tokens_4.input_ids:
    print(item)

print("Attention Mask")
for item in output_tokens_4.attention_mask:
    print(item)
```

```
Truncation was not explicitly activated but `max_length` is provided a specific value,␣
↪please use `truncation=True` to explicitly truncate examples to max length. Defaulting␣
↪to 'longest_first' truncation strategy. If you encode pairs of sequences (GLUE-style)␣
↪with the tokenizer you can select this strategy more precisely by providing a specific␣
↪strategy to `truncation`.
```

```
Input IDs
[101, 1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662, 102]
[101, 1045, 5223, 2023, 2061, 2172, 999, 102]
Attention Mask
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1]
```

The Tokenizers library also allows to train new tokenizers from scratch. For instance, if a large corpus of text is available in another language than English, a new tokenizer will need to be trained to efficiently deal with the differences in the punctuation and use of spaces in that language.

7.21.5 21.5 Datasets

Hugging Face provides access to a large number of **datasets**. If you wish to check all datasets please follow this link. As well as, to see all available datasets in Hugging Face, we can import `list_datasets`.

[ ]:
```
from datasets import list_datasets

all_datasets = list_datasets()
```

```
<ipython-input-37-cac6c310c34b>:3: FutureWarning: list_datasets is deprecated and will␣
↪be removed in the next major version of datasets. Use 'huggingface_hub.list_datasets'␣
↪instead.
  all_datasets = list_datasets()
```

[ ]:
```
print('Number of datasets in Hugging Face:', len(all_datasets))
```

```
Number of datasets in Hugging Face: 75567
```

[ ]:
```
print('Display the first 10 datasets:', '\n'.join(all_datasets[:10]))
```

```
Display the first 10 datasets: acronym_identification
ade_corpus_v2
adversarial_qa
aeslc
afrikaans_ner_corpus
ag_news
ai2_arc
air_dialogue
ajgt_twitter_ar
allegro_reviews
```

Let's load the Emotions dataset. It contains Twitter messages with six basic emotions: anger, fear, joy, love, sadness, and surprise. We can just use `load_dataset()` to accomplish that.

[ ]:
```
from datasets import load_dataset

emotions = load_dataset("emotion")
```

```
Downloading builder script:   0%|          | 0.00/3.97k [00:00<?, ?B/s]
Downloading metadata:   0%|          | 0.00/3.28k [00:00<?, ?B/s]
Downloading readme:   0%|          | 0.00/8.78k [00:00<?, ?B/s]
Downloading data files:   0%|          | 0/3 [00:00<?, ?it/s]
Downloading data:   0%|          | 0.00/592k [00:00<?, ?B/s]
Downloading data:   0%|          | 0.00/74.0k [00:00<?, ?B/s]
Downloading data:   0%|          | 0.00/74.9k [00:00<?, ?B/s]
Extracting data files:   0%|          | 0/3 [00:00<?, ?it/s]
Generating train split:   0%|          | 0/16000 [00:00<?, ? examples/s]
Generating validation split:   0%|          | 0/2000 [00:00<?, ? examples/s]
Generating test split:   0%|          | 0/2000 [00:00<?, ? examples/s]
```

We can see in the next cell that `emotions` dataset is a dictionary object that is split into training, validation, and test data sets, consisting of 16,000, 2,000, and 2,000 messages, respectively.

[ ]:
```
emotions
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 16000
    })
    validation: Dataset({
        features: ['text', 'label'],
        num_rows: 2000
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 2000
    })
})
```

The first and second training samples are shown next. They contain the text and the corresponding emotion label.

[ ]:
```
emotions['train'][0]
```

```
{'text': 'i didnt feel humiliated', 'label': 0}
```

[ ]:
```
emotions['train'][1]
```

```
{'text': 'i can go from feeling so hopeless to so damned hopeful just from being around␣
↪someone who cares and is awake',
 'label': 0}
```

The order of the labels for the emotion categories are shown in the next cell.

[ ]:
```
emotions['train'].features
```

```
{'text': Value(dtype='string', id=None),
 'label': ClassLabel(names=['sadness', 'joy', 'love', 'anger', 'fear', 'surprise'],␣
↪id=None)}
```

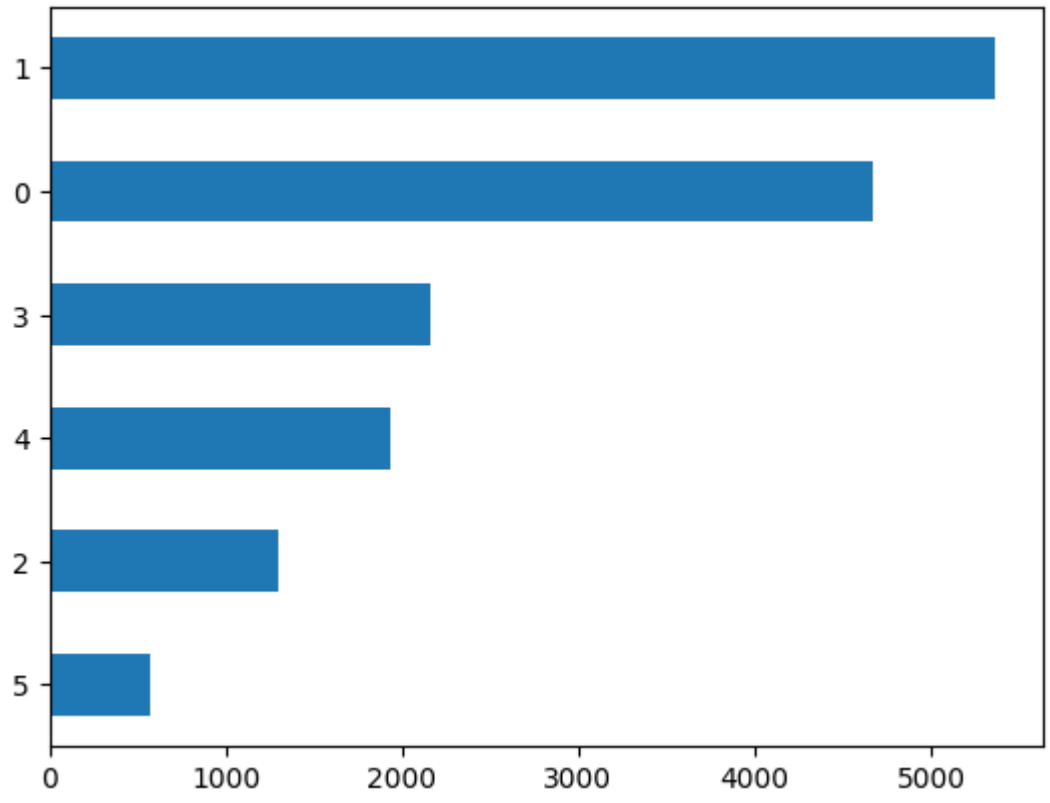
Hugging Face also allows to use the `set_format()` method with datasets, and to define the format of the data. For instance, by setting the type to Pandas, we can obtain the data as Pandas DataFrames.

[ ]:
```
emotions.set_format(type='pandas')
df = emotions["train"][:]
df.head()
```

```
                                                text  label
0                            i didnt feel humiliated      0
1  i can go from feeling so hopeless to so damned...      0
2   im grabbing a minute to post i feel greedy wrong      3
3  i am ever feeling nostalgic about the fireplac...      2
4                              i am feeling grouchy      3
```

We can use a bar plot to plot the number of values in each category.

[ ]:
```
import matplotlib.pyplot as plt

df["label"].value_counts(ascending=True).plot.barh()
plt.show()
```



Apply a Tokenizer to a Dataset

Let's show next how we can use a tokenizer to convert the first 5 samples in the training dataset to sequences of tokens.

[ ]:
```
emotions.set_format(type=None)

training_samples_5 = emotions["train"][:5]
training_samples_5
```

```
{'text': ['i didnt feel humiliated',
  'i can go from feeling so hopeless to so damned hopeful just from being around someone␣
↪who cares and is awake',
  'im grabbing a minute to post i feel greedy wrong',
  'i am ever feeling nostalgic about the fireplace i will know that it is still on the␣
↪property',
  'i am feeling grouchy'],
 'label': [0, 0, 3, 2, 3]}
```

Let's extract only the text data, and not the labels.

[ ]:
```
text_training_5 = training_samples_5['text']
text_training_5
```

```
['i didnt feel humiliated',
 'i can go from feeling so hopeless to so damned hopeful just from being around someone␣
↪who cares and is awake',
 'im grabbing a minute to post i feel greedy wrong',
 'i am ever feeling nostalgic about the fireplace i will know that it is still on the␣
↪property',
 'i am feeling grouchy']
```

Apply the tokenizer, and display the sequence of tokens.

[ ]:
```
output_tokens_5 = tokenizer(text_training_5, padding=True)

print("Input IDs")
for item in output_tokens_5.input_ids:
    print(item)

print("Attention Mask")
for item in output_tokens_5.attention_mask:
    print(item)
```

```
Input IDs
[101, 1045, 2134, 2102, 2514, 26608, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[101, 1045, 2064, 2175, 2013, 3110, 2061, 20625, 2000, 2061, 9636, 17772, 2074, 2013,␣
↪2108, 2105, 2619, 2040, 14977, 1998, 2003, 8300, 102]
[101, 10047, 9775, 1037, 3371, 2000, 2695, 1045, 2514, 20505, 3308, 102, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0]
[101, 1045, 2572, 2412, 3110, 16839, 9080, 12863, 2055, 1996, 13788, 1045, 2097, 2113,␣
↪2008, 2009, 2003, 2145, 2006, 1996, 3200, 102, 0]
[101, 1045, 2572, 3110, 24665, 7140, 11714, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0]
Attention Mask
[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

(continues on next page)

(continued from previous page)

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
[1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

7.21.6 21.6 Models

Importing a Pretrained Model

Instead of using a default model for a task, we can also select a language model from the many available in Hugging Face. Let's use GPT-2 language model, and we will also need to use the corresponding tokenizer for GPT-2.

The parameter `pad_token_id` is optional, and its purpose is to define the ID (assigned integer value) for the token used for padding the text sequences. In this case, the ID of the padding token is set to the ID of the end-of-sequence token (`eos_token_id`), which is a common choice for padding in language models.

[ ]:
```
from transformers import GPT2LMHeadModel
from transformers import GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")

model = GPT2LMHeadModel.from_pretrained("gpt2", pad_token_id = tokenizer.eos_token_id)
```

Now, let's consider the following sentence `input_string`, and first preprocess it with the tokenizer to `encode` it into a sequence of integers. The argument `return_tensors="pt"` specifies that the output should be returned as PyTorch tensors, where `"pt"` stands for PyTorch.

Next, we will use the GPT-2 model for text generation, to continue the input sentences. The `decode` method will convert the generated sequence by the model into text, here named `output_string`.

Two common methods for generating text with Language Models are:

- **Greedy search**, selects the word with the highest probability as the next word. The major drawback of greedy search is that it can miss potentially high-probability words that follow a low-probability word. Therefore, although each individual word may be the best fit when generating a response, the entire generated text can be less relevant for the query.
- **Beam search**, selects a sequence of words (beam) that has the overall highest probability. This approach reduces the risk of missing high-probability words, because rather than focusing only on the next word in a sequence, beam search looks at the probability of the entire response.

Beam search is typically preferred than greedy search, because with beam search, the model can consider multiple routes and find the best option.

For example, in the following figure, the input query to the model is "The Financial Times is …". The model created four possible beams with a potential response, and out of the four beams, the third beam "a newspaper founded in 1888" was selected as the most coherent human-like response.

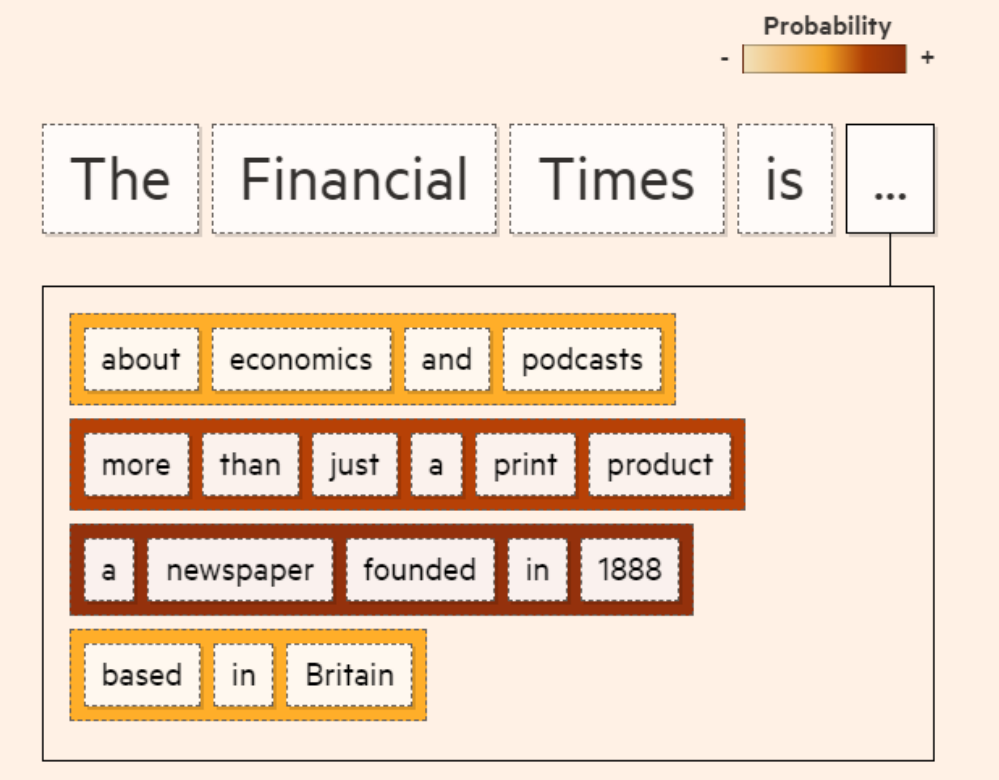


Figure: Beam search example. Source: link

[ ]:
```
input_string = "Yesterday I spent several hours in the library, studying"
input_tokens = tokenizer.encode(input_string, return_tensors = "pt")

output_greedy = model.generate(input_tokens, max_length=64)

output_string = tokenizer.decode(output_greedy[0], skip_special_tokens=True)
print(f"Output sequence: {output_string}")
```

```
Output sequence: Yesterday I spent several hours in the library, studying the books, and␣
↪I was amazed at how much I had learned. I was amazed at how much I had learned. I was␣
↪amazed at how much I had learned. I was amazed at how much I had learned. I was amazed␣
↪at how much I had learned.
```

You can notice that the generated text sequence with greedy search is not the best, since after the initial sequence of words, the model got stuck into a loop, and began repeating the same sentence. To deal with repetitive text generation we can apply a beam search strategy. Beam search examines multiple probable solutions (beams) for the generated text, which is set by the argument `num_beams=32`. We also applied a penalty term for repeating the same words. By considering severl possible solutions, the beam search algorithm tries to improve the generated output text by the model.

[ ]:
```
output_beam = model.generate(input_tokens, max_length=64, num_beams=32, \
       no_repeat_ngram_size=2, early_stopping=True)

output_string = tokenizer.decode(output_beam[0], skip_special_tokens=True)

print("Output sequence:\n", output_string)
```

```
Output sequence:
```

(continues on next page)

(continued from previous page)

```
 Yesterday I spent several hours in the library, studying all the books I could get my␣
↪hands on, and trying to figure out what I wanted to do with them. I didn't know what␣
↪to expect, but I did know that it was going to be a lot of fun.

When I got home, I
```

Fine-tuning a Pretrained Model to the Emotions Dataset

The next section demonstrates how to use a pretrained model and tokenizer in Hugging Face, and fine-tune the model to a dataset. We will use the DistilBERT model for this task, and we will again use the Emotions dataset.

Let's first download the tokenizer for the `"distilbert-base-uncased"` model.

[ ]:
```
model_ckpt = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
```

The following code tokenizes the emotions dataset. It is common to use the `Dataset.map()` method in Hugging Face for tokenization, which applies a function on each sample in the dataset. Therefore, we first define a function `tokenize` that is used in the `Dataset.map()` method. Using the option `batched=True` in the `map()` method will apply the tokenization to a batch of input sequences instead to each text sequence, which will speed up the tokenization. And, as we learned in the previous section, `tokenize` returns a dictionary with keys `input_ids` and `attention_mask`.

[ ]:
```
def tokenize(rows):
    return tokenizer(rows['text'], padding="max_length", truncation=True)

emotions.set_format(type=None)

tokenized_datasets = emotions.map(tokenize, batched=True)
```

```
Map:   0%|          | 0/16000 [00:00<?, ? examples/s]
```

```
Map:   0%|          | 0/2000 [00:00<?, ? examples/s]
```

```
Map:   0%|          | 0/2000 [00:00<?, ? examples/s]
```

[ ]:
```
tokenized_train_dataset = tokenized_datasets["train"]
tokenized_eval_dataset = tokenized_datasets["test"]
```

`DefaultDataCollator` in Hugging Face converts the data into appropriate format for model training. In this case we used `return_tensors="tf"`, therefore it will transform the dataset into TensorFlow type data, which we will use to train a neural network model defined in TensorFlow. I.e., the collate function will convert the tokenized input samples into TensorFlow tensors, and concatenate the tensors to create batches for training the model.

[ ]:
```
from transformers import DefaultDataCollator

data_collator = DefaultDataCollator(return_tensors="tf")
```

The train and validation datasets are defined in the next cell, where the `columns` and `label_col` define the text and labels in the datasets.

[ ]:
```
tf_train_dataset = tokenized_train_dataset.to_tf_dataset(
    columns=["attention_mask", "input_ids"],
    label_cols=["label"],
```

(continues on next page)

(continued from previous page)

```
    shuffle=True,
    collate_fn=data_collator,
    batch_size=8)

tf_validation_dataset = tokenized_eval_dataset.to_tf_dataset(
    columns=["attention_mask", "input_ids"],
    label_cols=["label"],
    shuffle=False,
    collate_fn=data_collator,
    batch_size=8)
```

```
/usr/local/lib/python3.10/dist-packages/datasets/arrow_dataset.py:400: FutureWarning:␣
↪The output of `to_tf_dataset` will change when a passing single element list for␣
↪`labels` or `columns` in the next datasets version. To return a tuple structure rather␣
↪than dict, pass a single string.
Old behaviour: columns=['a'], labels=['labels'] -> (tf.Tensor, tf.Tensor)
             : columns='a', labels='labels' -> (tf.Tensor, tf.Tensor)
New behaviour: columns=['a'],labels=['labels'] -> ({'a': tf.Tensor}, {'labels': tf.
↪Tensor})
             : columns='a', labels='labels' -> (tf.Tensor, tf.Tensor)
  warnings.warn(
```

Next, we will load the DistilBERT model for classification. Hugging Face Transformers library provides an `AutoModel` class which also has a `from_pretrained()` method that can be used to load a pretrained checkpoint. Hugging Face offers several types of AutoModel instances, and since we will need a model to perform classification of Twitter messages into classes, and the dataset is preprocessed into TesorFlow data type, we will use the model `TFAutoModelForSequenceClassification`.

After the model is defined, we will compile it and re-train using our dataset.

[ ]:
```
import tensorflow as tf
from transformers import TFAutoModelForSequenceClassification

model = TFAutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased",␣
↪num_labels=6)
```

```
Downloading model.safetensors:   0%|          | 0.00/268M [00:00<?, ?B/s]
```

```
Some weights of the PyTorch model were not used when initializing the TF 2.0 model␣
↪TFDistilBertForSequenceClassification: ['vocab_layer_norm.bias', 'vocab_transform.bias
↪', 'vocab_transform.weight', 'vocab_layer_norm.weight', 'vocab_projector.bias']
- This IS expected if you are initializing TFDistilBertForSequenceClassification from a␣
↪PyTorch model trained on another task or with another architecture (e.g. initializing␣
↪a TFBertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing TFDistilBertForSequenceClassification␣
↪from a PyTorch model that you expect to be exactly identical (e.g. initializing a␣
↪TFBertForSequenceClassification model from a BertForSequenceClassification model).
Some weights or buffers of the TF 2.0 model TFDistilBertForSequenceClassification were␣
↪not initialized from the PyTorch model and are newly initialized: ['pre_classifier.
↪weight', 'pre_classifier.bias', 'classifier.weight', 'classifier.bias']
You should probably TRAIN this model on a down-stream task to be able to use it for␣
↪predictions and inference.
```

[ ]:
```
from tensorflow import keras
from keras.optimizers import Adam
from keras.losses import SparseCategoricalCrossentropy
from keras.metrics import SparseCategoricalAccuracy
```

[ ]:
```
model.compile(optimizer=Adam(learning_rate=5e-5),
    loss=SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy'])
```

[ ]:
```
model.fit(tf_train_dataset, validation_data=tf_validation_dataset, epochs=5)
```

```
Epoch 1/5
2000/2000 [==============================] - 979s 476ms/step - loss: 0.3840 - sparse_
↪categorical_accuracy: 0.8656 - val_loss: 0.1661 - val_sparse_categorical_accuracy: 0.
↪9210
Epoch 2/5
2000/2000 [==============================] - 960s 480ms/step - loss: 0.1546 - sparse_
↪categorical_accuracy: 0.9324 - val_loss: 0.1549 - val_sparse_categorical_accuracy: 0.
↪9220
Epoch 3/5
2000/2000 [==============================] - 964s 482ms/step - loss: 0.1249 - sparse_
↪categorical_accuracy: 0.9382 - val_loss: 0.2141 - val_sparse_categorical_accuracy: 0.
↪9135
Epoch 4/5
2000/2000 [==============================] - 970s 485ms/step - loss: 0.1128 - sparse_
↪categorical_accuracy: 0.9495 - val_loss: 0.1692 - val_sparse_categorical_accuracy: 0.
↪9150
Epoch 5/5
2000/2000 [==============================] - 961s 481ms/step - loss: 0.0974 - sparse_
↪categorical_accuracy: 0.9580 - val_loss: 0.1974 - val_sparse_categorical_accuracy: 0.
↪9235
```

```
<keras.src.callbacks.History at 0x7fbb843b27a0>
```

7.21.7 References

1. Hugging Face Course, available at https://huggingface.co/course/chapter1/1.
2. Applications of Deep Neural Networks, Course at Washington University in St. Louis, Jeff Heaton, available at https://github.com/jeffheaton/t81_558_deep_learning/blob/master/t81_558_class_11_01_huggingface.ipynb.
3. Getting Started with Hugging Face Transformers for NLP, Exxact Blog, available at https://www.exxactcorp.com/blog/Deep-Learning/getting-started-hugging-face-transformers.
4. An Introduction to Using Transformers and Hugging Face, Zoumana Kelta, available at https://www.datacamp.com/tutorial/an-introduction-to-using-transformers-and-hugging-face.

BACK TO TOP

7.22 Lecture 22 - Diffusion Models for Text-to-Image Generation

7.22.1 22.1 Generative Text-to-Image Models

Generative models are a category of Machine Learning models that generate new data instances, typically based on the learned data distribution of the training dataset. This is different from most models that we studied so far, such as *discriminative models* that can be used to classify data instances into several classes, for example.

The family of GAN models (StyleGAN, CycleGAN, BigGAN, etc.) have been the most important category of generative models since the original GAN paper was published in 2014. Despite the remarkable progress in image synthesis achieved by the family of GAN models, these models are generally difficult to train due to their adversarial nature, as well as there is a lack of diversity in generated images.

Another group of generative models were recently been introduced, referred to as **Denoising Diffusion Probabilistic Models** (DDPMs), or they are also simply called Diffusion Models. Diffusion Models convert Gaussian random noise into images from a learned data distribution in an iterative denoising process. This approach is inspired by the physical process of gas diffusion, and has also applications in other scientific fields. Diffusion Models have demonstrated an ability to generate images with increased quality and diversity in comparison to GANs, and they don't suffer from mode collapse and other training instabilities that are characteristical for GANs.

Examples of recent text-to-image generative methods that are based on Diffusion Models include DALL-E 2 by OpenAI, Stable Diffusion by Stability.AI, and ImageGen by Google Brain.

Text-to-image generative models rely on learned representations by Large Language Models that are trained on pairs of images and image captions. These representations are afterward used to guide a Diffusion Model in generating new images given a text prompt.

This group of generative models can also take an input image as a prompt, and generate an image similar to the input image. For instance, we can quickly draw a simple drawing manually, and use it to create a photorealistic image.

7.22.2 22.2 High-Level API for Stable Diffusion with Keras

Let's first show an example of a high-level API for generating images from text prompts, and afterward we will explain the components in text-to-image models.

The code in the next cells employs a Keras library for Computer Vision called `keras_cv` that implements the Stable Diffusion model. Stable Diffusion was the first text-to-image model that was open-sourced, and this motivated a large number of recent applications.

To run the model we need to first install the `keras_cv` package. Afterward, we can just load a Stable Diffusion model and give a prompt to generate images.

[ ]:
```
!pip install keras_cv --upgrade --quiet
```

```
     803.1/803.1 kB 10.8 MB/s eta 0:00:00
     950.8/950.8 kB 48.5 MB/s eta 0:00:00
```

[ ]:
```
import keras_cv
from tensorflow import keras
import matplotlib.pyplot as plt
```

```
Using TensorFlow backend
```

[ ]:
```
# instantiate a Stable Diffusion model
model = keras_cv.models.StableDiffusion(img_width=512, img_height=512)
```

```
By using this model checkpoint, you acknowledge that its usage is subject to the terms␣
↪of the CreativeML Open RAIL-M license at https://raw.githubusercontent.com/CompVis/
↪stable-diffusion/main/LICENSE
```

[ ]:
```
# generate images (it takes a few minutes with Google Colab Pro)
images = model.text_to_image("photograph of an astronaut riding a horse", batch_size=3)

# plot the images
plt.figure(figsize=(16, 8))
for i in range(len(images)):
    ax = plt.subplot(1, len(images), i + 1)
    plt.imshow(images[i])
    plt.axis("off")
```

```
Downloading data from https://github.com/openai/CLIP/blob/main/clip/bpe_simple_vocab_
↪16e6.txt.gz?raw=true
1356917/1356917 [==============================] - 0s 0us/step
Downloading data from https://huggingface.co/fchollet/stable-diffusion/resolve/main/kcv_
↪encoder.h5
492466864/492466864 [==============================] - 2s 0us/step
Downloading data from https://huggingface.co/fchollet/stable-diffusion/resolve/main/kcv_
↪diffusion_model.h5
3439090152/3439090152 [==============================] - 14s 0us/step
50/50 [==============================] - 176s 2s/step
Downloading data from https://huggingface.co/fchollet/stable-diffusion/resolve/main/kcv_
↪decoder.h5
198180272/198180272 [==============================] - 4s 0us/step
```

[ ]:
```
# generate images (it takes a few minutes per image with Google Colab Pro)
images = model.text_to_image("ruins of an ancient castle in a desert, concept art",␣
↪batch_size=3)

# plot the images
plt.figure(figsize=(16, 8))
for i in range(len(images)):
    ax = plt.subplot(1, len(images), i + 1)
    plt.imshow(images[i])
    plt.axis("off")
```

```
50/50 [==============================] - 95s 2s/step
```

[ ]:
```
# generate images
images = model.text_to_image("a cute magical flying dog, fantasy art, golden color, high␣
↪quality, highly detailed, elegant, sharp focus,"
    "concept art, character concepts, digital painting, mystery, adventure",  batch_
↪size=3)

# plot the images
plt.figure(figsize=(16, 8))
for i in range(len(images)):
    ax = plt.subplot(1, len(images), i + 1)
    plt.imshow(images[i])
    plt.axis("off")
```

```
50/50 [==============================] - 95s 2s/step
```

7.22.3 22.3 Denoising Diffusion Probabilistic Models

Denoising diffusion probabilistic process consists of two phases:

1. *Forward diffusion process*, in which Gaussian noise is gradually added to an image, until the image becomes complete random noise.
2. *Reverse diffusion process*, in which the image with Gaussian noise is gradually denoised, until all noise is removed and the original image is recovered.

22.3.1 Forward Diffusion Process

For an initial image denoted x_0 that is sampled from the data distribution $q(x)$, the forward diffusion process adds noise over T consecutive steps. At each step, the added Gaussian noise has variance β_t. I.e., the sample x_t that corresponds to step t of the forward process is obtained by adding Gaussian noise to the sample x_{t-1} from step $t1$. Therefore the conditional probability density of the forward diffusion process $q(x_t|x_{t-1})$ can be written as:

$$q(x_t \mid x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t\mathbf{I})$$

The above notation means that x_t has Gaussian distribution, with a mean $\sqrt{1-\beta_t}x_{t-1}$ and variance $\beta_t\mathbf{I}$.

This process is depicted in the next figure.

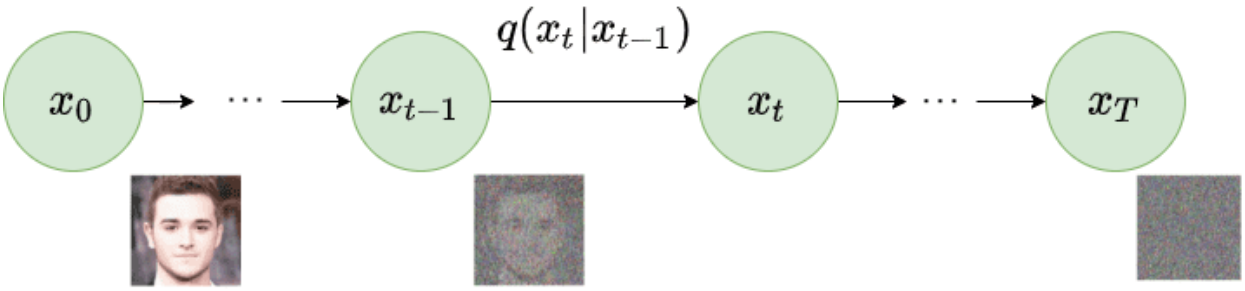


Figure: Forward diffusion process.

Such process where the probability of an event at any given state is dependent only on the immediately preceding state is called *Markov Chain* or *Markov Process*.

The mean and variance of the samples depend on a parameter β_t, which defines the level of added Gaussian noise. The value of β_t can either be constant for all steps, or it can be gradually changed (e.g., by using a sigmoid, cosine, tanh, linear function, etc).

Substituting $\alpha_t = 1 - \beta_t$, and $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$, the distribution can also be rewritten as:

$$q(x_t \mid x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t} x_0, (1 - \bar{\alpha}_t)\mathbf{I})$$

This substitution allows to draw samples at any timestep by using only the initial image x_0, without going through the intermediate steps. Hence, the variable x_t of the forward diffusion can be written in terms of the initial image x_0 and the random Gaussian noise $\epsilon \sim \mathcal{N}(0, 1)$ as:

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon$$

Note that the forward diffusion process does not involve learning. It is a simple process that just applies noise to an image.

Example of Forward Diffusion

An example of performing forward diffusion is shown below, based on the code repository at https://github.com/azad-academy/denoising-diffusion-model.

The codes in the next cells apply Gaussian noise to an image of the letter S over 100 steps. In the figure below, the images for every 10th step of the noise-adding process are shown.

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
cd 'drive/MyDrive/Data_Science_Course/Fall_2023/Lecture_22-Diffusion_Models/helper_
↪functions/'
```

```
/content/drive/MyDrive/Data_Science_Course/Fall_2023/Lecture_22-Diffusion_Models/helper_
↪functions
```

[ ]:
```
# code from: https://github.com/azad-academy/denoising-diffusion-model

### Note: the code in the next cells for this example is not required for quizzes or␣
↪assignments  ###

import numpy as np
from sklearn.datasets import make_s_curve
from helper_plot import hdr_plot_style
import torch
from utils import *

# create S curve
s_curve, _= make_s_curve(10**4, noise=0.1)
s_curve = s_curve[:, [0, 2]]/10.0

data = s_curve.T

# plot the S curve
hdr_plot_style()
plt.scatter(*data, alpha=0.5, color='white', edgecolor='gray', s=5)
plt.axis('off')

dataset = torch.Tensor(data.T).float()
```

[ ]:
```
num_steps = 100

# apply a sigmoid schedule for the beta parameter
betas = make_beta_schedule(schedule='sigmoid', n_timesteps=num_steps, start=1e-5, end=0.
↪5e-2)

# substitute alphas
alphas = 1 - betas
alphas_prod = torch.cumprod(alphas, 0)
alphas_prod_p = torch.cat([torch.tensor([1]).float(), alphas_prod[:-1]], 0)
alphas_bar_sqrt = torch.sqrt(alphas_prod)
one_minus_alphas_bar_log = torch.log(1 - alphas_prod)
one_minus_alphas_bar_sqrt = torch.sqrt(1 - alphas_prod)
```

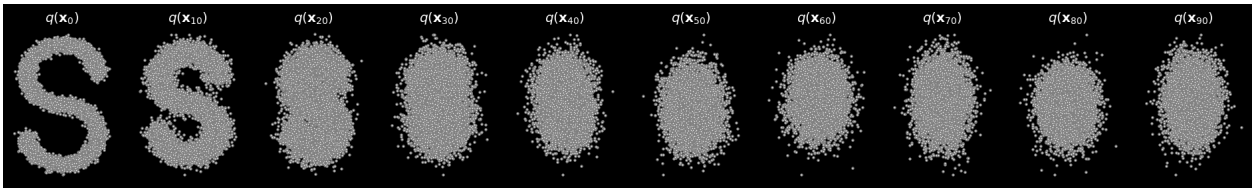
[ ]:
```
# draw a sample at step t, given an initial image x_0
def q_x(x_0, t, noise=None):
    if noise is None:
        noise = torch.randn_like(x_0)
    alphas_t = extract(alphas_bar_sqrt, t, x_0)
    alphas_1_m_t = extract(one_minus_alphas_bar_sqrt, t, x_0)
    return (alphas_t * x_0 + alphas_1_m_t * noise)
```

[ ]:
```
# plot the samples for steps 0, 10, 20, ..., 90
fig, axs = plt.subplots(1, 10, figsize=(24, 3))
for i in range(10):
    q_i = q_x(dataset, torch.tensor([i * 10]))
```

(continues on next page)

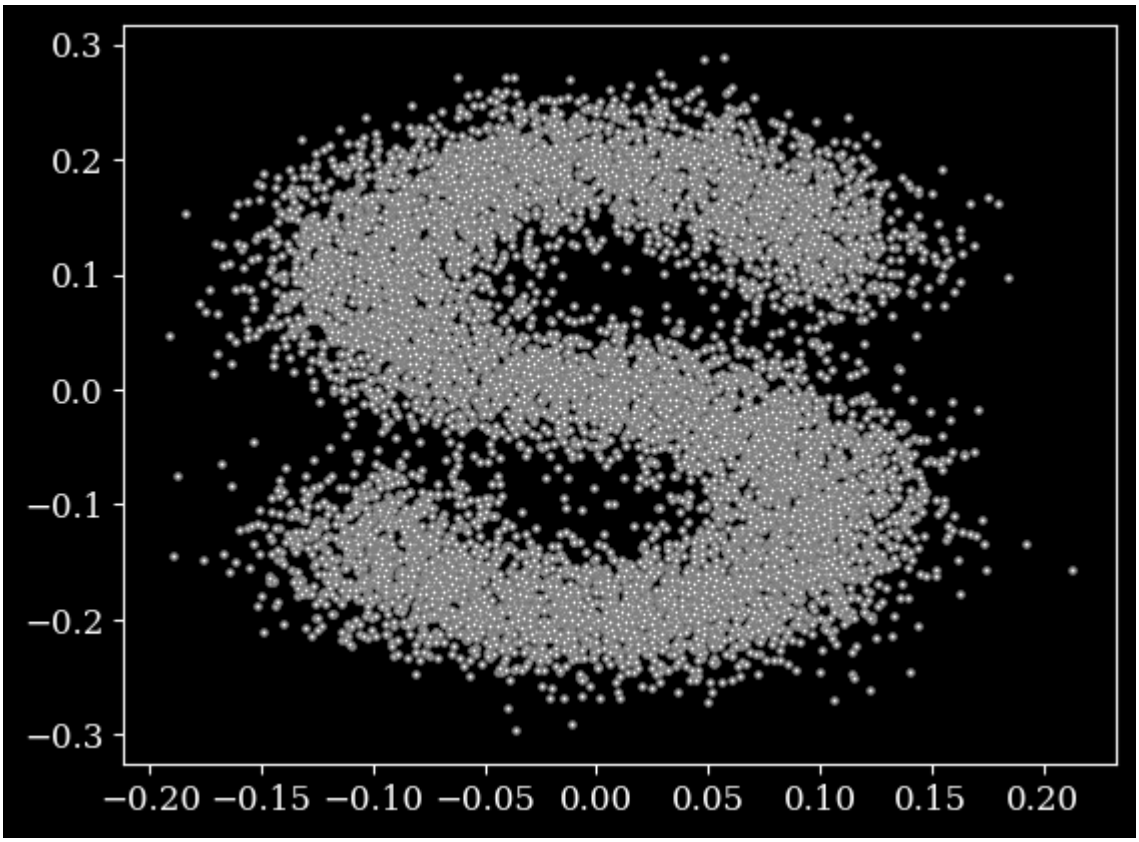
(continued from previous page)

```
    axs[i].scatter(q_i[:, 0], q_i[:, 1], color='white', edgecolor='gray', s=5)
    axs[i].set_axis_off(); axs[i].set_title('$q(\mathbf{x}_'+str(i*10)+')$')
```



As mentioned above, we can obtain a sample at any desired step in the forward diffusion process, by applying the level of Gaussian noise directly to the initial image. The cell below shows the image at step 15.

[ ]:
```
# image in the 15th step
q_15 = q_x(dataset, torch.tensor([15]))
plt.scatter(q_15[:, 0], q_15[:, 1], color='white', edgecolor='gray', s=5);
```



22.3.2 Reverse Diffusion Process

The goal of the reverse diffusion process is to denoise the images from the forward diffusion process, i.e., start with a noisy image and obtain a clean image. Accurately calculating the reverse process $q(x_{t-1}|x_t)$ is intractable, and therefore, text-to-image models apply deep neural networks to approximate the probability density function $q(x_{t-1}|x_t)$ with a parameterized model $p_\theta(x_{t-1}|x_t)$, where θ denotes the parameters of the deep neural network model that are learned. The model takes as input a noisy image at step x_t and predicts the mean $\mu_\theta(x_t, t)$ and the variance $\Sigma_\theta(x_t, t)$ of a denoised image x_{t-1}.

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t))$$

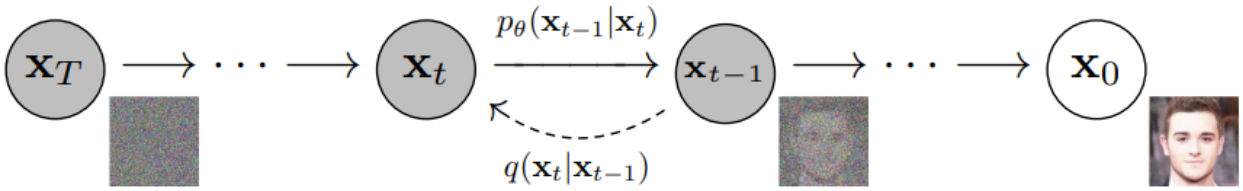


Figure: Reverse diffusion process.

Example of Reverse Diffusion

The reverse diffusion process for the image S from the above example is presented below. The mathematical expressions for calculating the posteriors are omitted here, and can be found in reference [3]. The model is trained for 1,000 iterations, and the denoising images at steps 0, 250, 500, 750, and 1,000 are shown in the figure below. In the last row of images, we can see that the model begins with a noisy image on the left and it ends with a denoised image on the right.

[ ]:
```
posterior_mean_coef_1 = (betas * torch.sqrt(alphas_prod_p) / (1 - alphas_prod))
posterior_mean_coef_2 = ((1 - alphas_prod_p) * torch.sqrt(alphas) / (1 - alphas_prod))
posterior_variance = betas * (1 - alphas_prod_p) / (1 - alphas_prod)
posterior_log_variance_clipped = torch.log(torch.cat((posterior_variance[1].view(1, 1),␣
↪posterior_variance[1:].view(-1, 1)), 0)).view(-1)

def q_posterior_mean_variance(x_0, x_t, t):
    coef_1 = extract(posterior_mean_coef_1, t, x_0)
    coef_2 = extract(posterior_mean_coef_2, t, x_0)
    mean = coef_1 * x_0 + coef_2 * x_t
    var = extract(posterior_log_variance_clipped, t, x_0)
    return mean, var
```

The U-Net model is imported in the next cell, and the model is trained for 1,000 iterations.

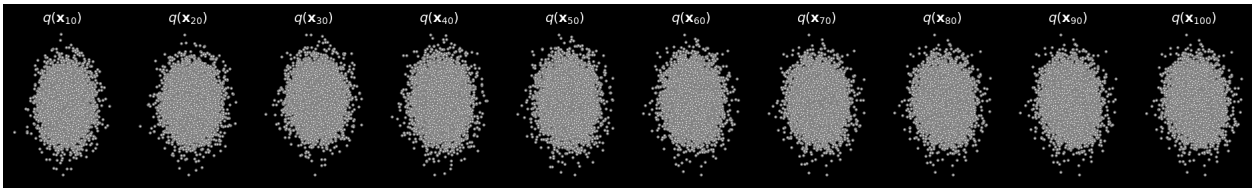
[ ]:
```
from model import ConditionalModel
from ema import EMA
import torch.optim as optim

model = ConditionalModel(num_steps)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
# Create EMA model
ema = EMA(0.9)
```

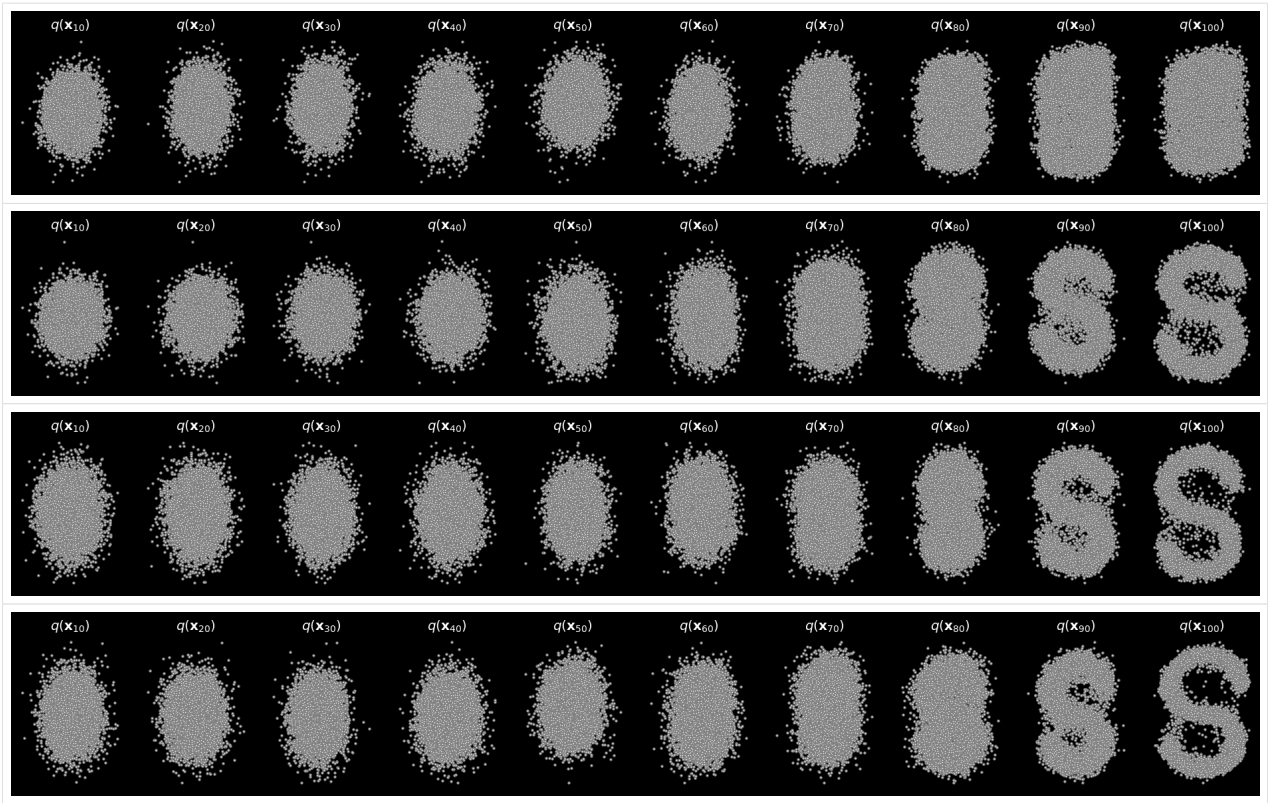
(continues on next page)

(continued from previous page)

```
ema.register(model)
# Batch size
batch_size = 128

# training
for t in range(1000):
    # X is a torch Variable
    permutation = torch.randperm(dataset.size()[0])
    for i in range(0, dataset.size()[0], batch_size):
        # Retrieve current batch
        indices = permutation[i:i+batch_size]
        batch_x = dataset[indices]
        # Compute the loss
        loss = noise_estimation_loss(model, batch_x,alphas_bar_sqrt,one_minus_alphas_bar_
↪sqrt,num_steps)
        # Before the backward pass, zero all of the network gradients
        optimizer.zero_grad()
        # Backward pass: compute gradient of the loss with respect to parameters
        loss.backward()
        # Perform gradient clipping
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.)
        # Calling the step function to update the parameters
        optimizer.step()
        # Update the exponential moving average
        ema.update(model)
    # Print loss
    if (t % 250==0) or (t==999):
        print('Iteration:', t)
        x_seq = p_sample_loop(model, dataset.shape,num_steps,alphas,betas,one_minus_
↪alphas_bar_sqrt)
        fig, axs = plt.subplots(1, 10, figsize=(24, 3))
        for i in range(1, 11):
            cur_x = x_seq[i * 10].detach()
            axs[i-1].scatter(cur_x[:, 0], cur_x[:, 1],color='white',edgecolor='gray',␣
↪s=5);
            axs[i-1].set_axis_off();
            axs[i-1].set_title('$q(\mathbf{x}_{'+str(i*10)+'})$')
```

```
Iteration: 0
Iteration: 250
Iteration: 500
Iteration: 750
Iteration: 999
```





The gif image of the reverse diffusion process is shown below.

Figure: Animation of the reverse diffusion process.

U-Net Model for Reverse Diffusion

A U-Net model is used for the reverse diffusion phase. **U-Net** is a popular deep learning network for image segmentation. The name is due to the architecture that looks like the letter U. U-Net includes an encoder sub-network that extracts lower-dimensional representations, and a decoder sub-network that reconstructs the representations to full size images. I.e., the encoder first downsamples the input image (reducing its size), and afterward the decoder upsamples the representations to the size of the original image. Another important part of U-Net are the skip connections which connect the feature maps from the encoder to the decoder, and help with the gradient flow. When U-Net is used for image segmentation, the inputs are images, and the outputs are segmentation masks that segment the objects in the input images.

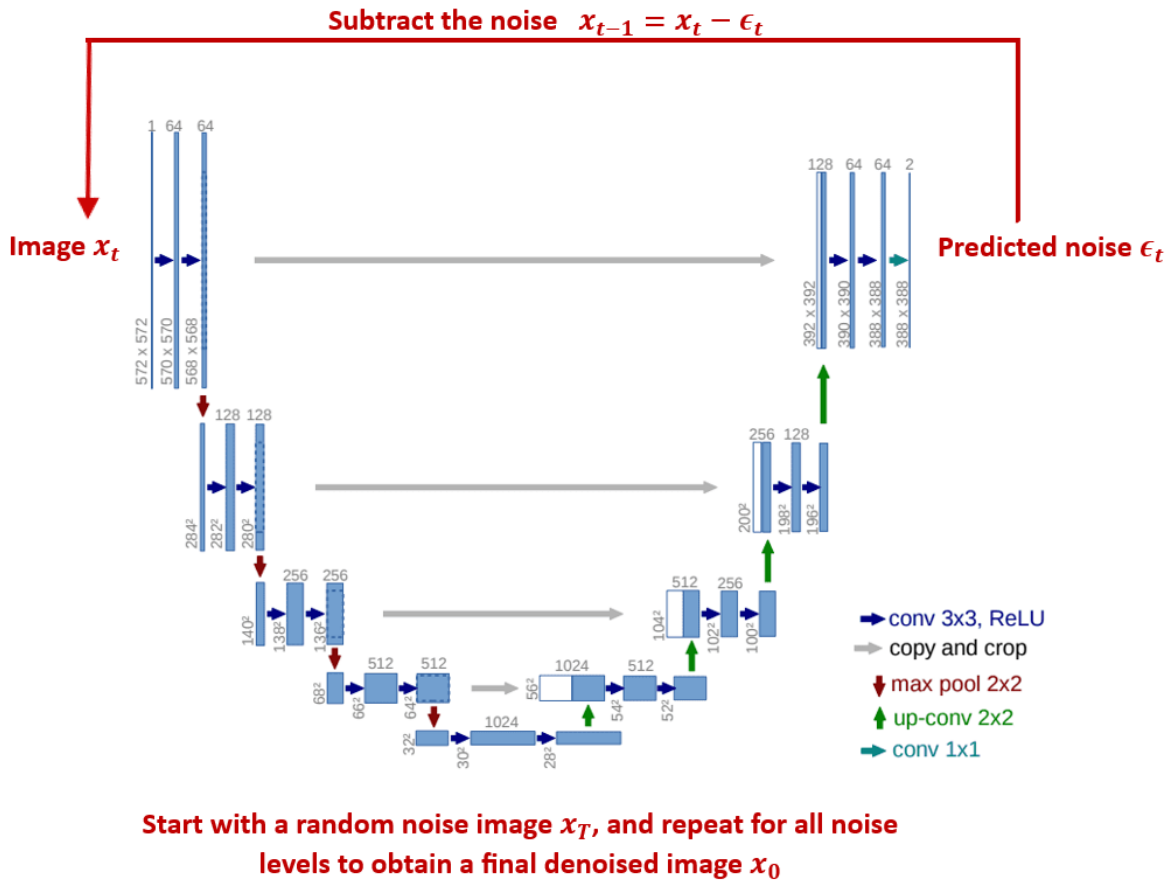


Figure: U-Net network.

In diffusion models, input to U-Net is a noisy image x_t at a particular time step t, and output by the model is the noise that has been added to the image in the previous time step of the forward diffusion process. By subtracting the predicted noise ϵ_t from the input image x_t, the result is a denoised image x_{t-1} at the previous time step. By repeating this process for every time step t from the final step T to the initial step 0, the model learns how to gradually create slightly less denoised images at each step. The final output for time step 0 is a fully denoised image x_0.

7.22.4 22.4 Text Encoder

To generate images based on a text prompt, text-to-image models employ text embeddings from a Language Model. In particular, the model CLIP (Contrastive Language Image Pretraining) has been used by several of these models.

CLIP employs a Transformer Network architecture. It is trained on a dataset of images and image captions, with an objective to match the text in image captions to the content in the corresponding images.

Figure: Images and captions dataset.

CLIP has an image encoder and a text encoder sub-network (shown in the figure). The two encoders convert the input pairs of images and image captions into image embeddings and text embeddings, respectively. During training, the model learns to group together image and text embeddings that belong to the same object. By repeating the training over a large dataset of images and image captions, the model updates the weights of the image and text encoders so that they output predictions that match the caption text and image content for the objects in the training dataset.

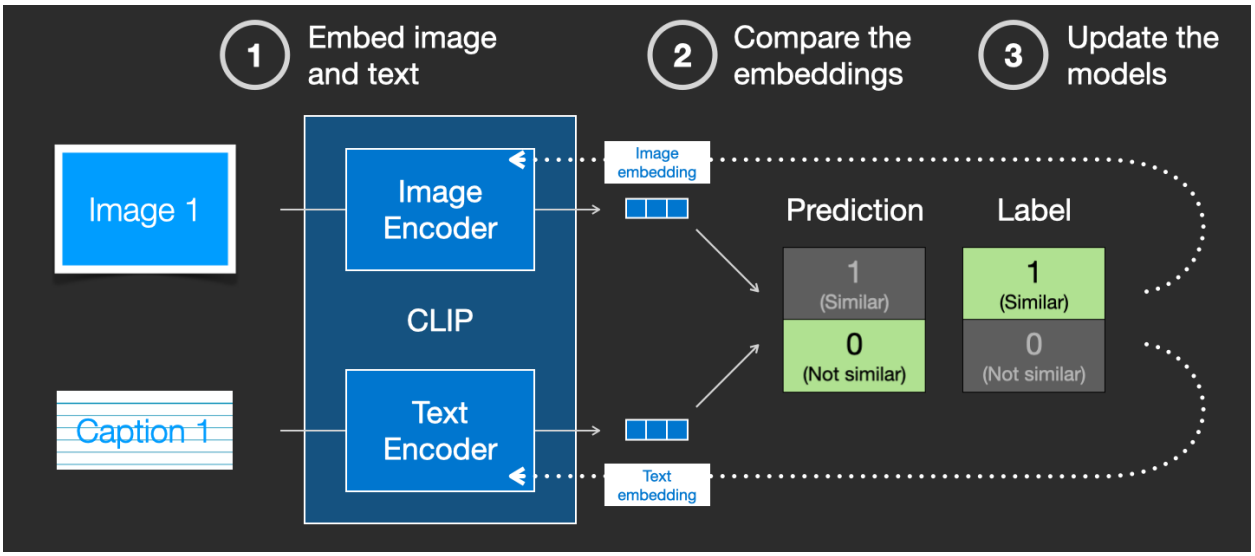


Figure: CLIP model.

7.22.5 22.5 Latent Diffusion Models

Latent Diffusion Models apply the diffusion process to a compressed image representation in a lower-dimensional space, instead of applying the diffusion process to the raw high-dimensional images. The lower-dimensional space is called *latent space*, hence the name for these models. Stable Diffusion is a Latent Diffusion Model.

The key advantage of Latent Diffusion Models is computational efficiency, due to processing small-size image representations. For instance, Stable Diffusion is trained on images of size 512x512 pixels, whereas the size of the image representations in the latent space is 64x64 pixels. Another advantage of performing the diffusion process in the latent space instead of the pixel space, is producing diverse images that preserve the semantic structure of the data.

An annotated figure of a Latent Diffusion Model is shown in the next figure. The components of the model are described next.

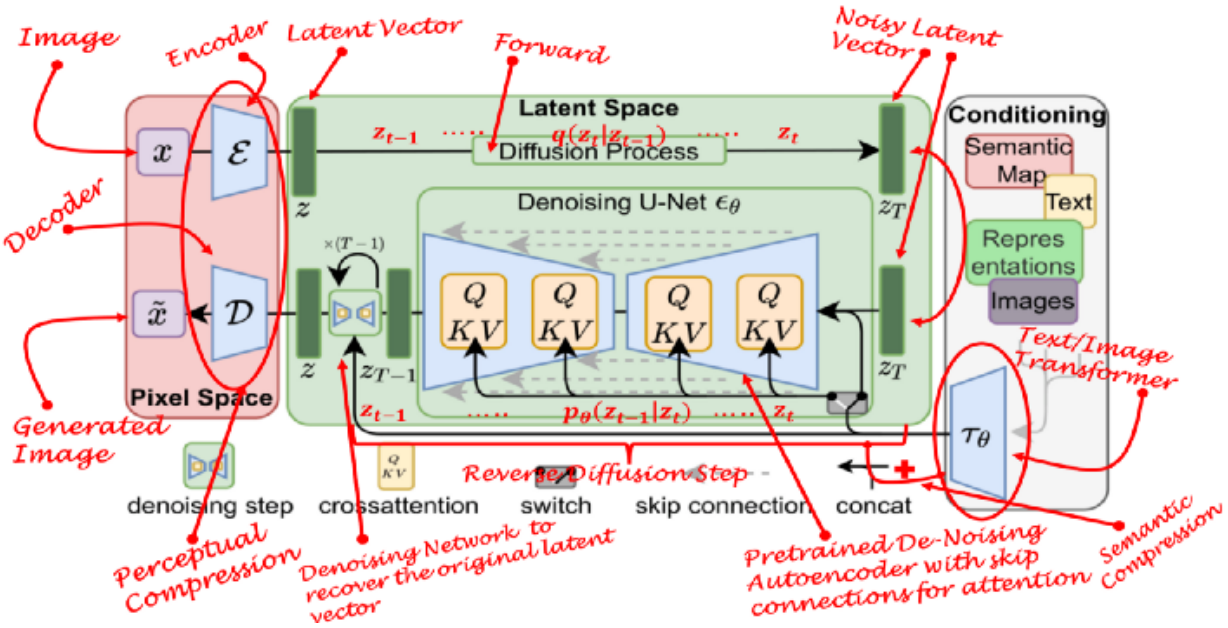


Figure: Latent diffusion model.

Perceptual Compression

The input image to the model x is shown in the upper left corner. The input image is in the pixel space, i.e., it consists of raw pixels. During the perceptual compression step, the image is projected into the latent space. For this step, an Encoder network $\mathcal{E}$ is employed to produce a lower-dimensional representation of the input image z that has smaller dimensions.

Forward Diffusion Step

Forward diffusion process is applied to the latent representation of the image z, by applying steps of gradually corrupting the image with Gaussian noise. The output of the forward diffusion step is a noise-corrupter image z_T.

Semantic Compression

This step corresponds to the right-hand block in the figure, and typically employs a Language Model to capture the semantic structure in text and images. An example is the use of the image-caption CLIP model from the previous section. This model encodes the text prompt by the user into a compressed representation, denoted τ_θ in the figure. This compressed representation of the user's prompt is used during the reverse diffusion step to control the visual content generated by the model and to guide the image generation process in order to ensure that the content in the denoised image corresponds to the text prompt.

Reverse Diffusion Step

The reverse diffusion step takes noisy images z_T and gradually removes the noise to generate clean images z. As we mentioned, a U-Net model is typically employed for learning the denoising process. The reverse diffusion process is *conditioned* on the text representation τ_θ from the pretrained CLIP transformer model. I.e., conditioning the reverse diffusion means that the model guides the denoising process by mapping the compressed text representations to the intermediate layers of the U-Net via cross-attention layers. The cross-attention layers are similar to the self-attention mechanism in Transformer Networks, and they learn a set of Q (queries), K (keys), and V (values) matrices. The U-Net network also has added positional encodings into each block, which specify the diffusion timestep t.

High-resolution Image Decoder

The obtained latent vector z is finally passed through a Decoder network $\mathcal{D}$ to increase the resolution back to the size of the original image. I.e., the resulting image representation in the latent space is 64x64 pixels, and the decoder recovers it to a high-resolution image with 512x512 pixels size in the pixel space. The generated image is denoted $\tilde{x}$ in the figure.

7.22.6 22.6 Generating Images with Stable Diffusion

The Stable Diffusion model trained by Stability.AI is available at Hugging Face. Let's use this model to generate images. Note that in the above code in Section 22.2 we used an implementation of Stable Diffusion by Keras.

To use this model requires to login at Hugging Face and obtain a token. After that, you can simply download the diffusers and transformers packages, and provide text prompts to generate images.

[ ]:
```
!pip install -q huggingface-hub

from huggingface_hub import notebook_login
notebook_login()
```

```
VBox(children=(HTML(value='<center> <img\nsrc=https://huggingface.co/front/assets/
↪huggingface_logo-noborder.sv...
```

[ ]:
```
!pip install -qq -U diffusers transformers accelerate
```

[ ]:
```
from diffusers import StableDiffusionPipeline
import torch

# set up the pipeline to generate images
pipe = StableDiffusionPipeline.from_pretrained("CompVis/stable-diffusion-v1-4", revision=
↪"fp16", torch_dtype=torch.float16).to('cuda')

prompt = "a photograph of an astronaut riding a horse"
# generate an image for the above prompt
pipe(prompt).images[0]
```

```
/usr/local/lib/python3.10/dist-packages/diffusers/pipelines/pipeline_utils.py:267:␣
↪FutureWarning: You are loading the variant fp16 from CompVis/stable-diffusion-v1-4 via␣
↪`revision='fp16'`. This behavior is deprecated and will be removed in diffusers v1.␣
↪One should use `variant='fp16'` instead. However, it appears that CompVis/stable-
↪diffusion-v1-4 currently does not have the required variant filenames in the 'main'␣
↪branch.
 The Diffusers team and community would be very grateful if you could open an issue:␣
↪https://github.com/huggingface/diffusers/issues/new with the title 'CompVis/stable-
↪diffusion-v1-4 is missing fp16 files' so that the correct variant file can be added.
```

(continues on next page)

(continued from previous page)

```
  warnings.warn(
vae/diffusion_pytorch_model.safetensors not found
```

```
Loading pipeline components...:   0%|          | 0/7 [00:00<?, ?it/s]
```

```
/usr/local/lib/python3.10/dist-packages/transformers/models/clip/feature_extraction_clip.
↪py:28: FutureWarning: The class CLIPFeatureExtractor is deprecated and will be removed␣
↪in version 5 of Transformers. Please use CLIPImageProcessor instead.
  warnings.warn(
`text_config_dict` is provided which will be used to initialize `CLIPTextConfig`. The␣
↪value `text_config["id2label"]` will be overriden.
`text_config_dict` is provided which will be used to initialize `CLIPTextConfig`. The␣
↪value `text_config["bos_token_id"]` will be overriden.
`text_config_dict` is provided which will be used to initialize `CLIPTextConfig`. The␣
↪value `text_config["eos_token_id"]` will be overriden.
```

```
  0%|          | 0/50 [00:00<?, ?it/s]
```

```
[ ]: prompt = "a cute magical flying dog, fantasy art, golden color, high quality, highly␣
↪detailed, elegant, sharp focus," \
    "concept art, character concepts, digital painting, mystery, adventure"
```

(continues on next page)

(continued from previous page)

```
# generate an image for the above prompt
pipe(prompt).images[0]
```

```
  0%|          | 0/50 [00:00<?, ?it/s]
```

The results from text-to-image models can vary based on the text prompt. *Prompt engineering* involves customizing the text prompts to influence the generated images. Examples of prompt engineering include specifying the painting style of the image type, lighting, focus, etc.

7.22.7 Appendix

The material in the Appendix is not required for quizzes and assignments.

Interpolation Between Prompts

Text-to-image models can be used to interpolate between the embedding representations of different text prompts. This is referred to as *latent space walking* or *latent space exploration* because the model samples data points in the latent space and incrementally changes the latent representation to reach from one text prompt to another text prompt.

One application is shown below, where each sampled point is saved as a frame, and a gif animation is created that shows the latent space walking between two prompts.

[ ]:
```
# Code from: https://github.com/svpino/stable-diffusion/blob/main/stable-diffusion.ipynb

PROMPT1 = "a ford truck"
PROMPT2 = "an electric car"

# number of interpolation steps between the two text prompts
INTERPOLATION_STEPS = 42

# number of frames per second
FRAMES_PER_SECOND = 10

# number of diffusion steps
DIFFUSION_STEPS = 25

# name of the file
ANIMATION_FILENAME = "animation.gif"

SEED = 42
```

[ ]:
```
import tensorflow as tf
from PIL import Image

# random noise patch
noise = tf.random.normal((512 // 8, 512 // 8, 4), seed=SEED)

# encoded vectors for the two text prompts
encoding1 = tf.squeeze(model.encode_text(PROMPT1))
encoding2 = tf.squeeze(model.encode_text(PROMPT2))

# generate the interpolation steps between the two encoded prompts
interpolated_encodings = tf.linspace(encoding1, encoding2, INTERPOLATION_STEPS)

# split the encodings into batches
batch_size = 3
batches = INTERPOLATION_STEPS // batch_size
encodings = tf.split(interpolated_encodings, batches)

animation = []
for batch in range(batches):
    # generate images
```

(continues on next page)

(continued from previous page)

```
    images = model.generate_image(
        encodings[batch],
        batch_size=batch_size,
        num_steps=DIFFUSION_STEPS,
        diffusion_noise=noise,
    )

    # add the images to the animation
    animation.extend(map(lambda image: Image.fromarray(image), images))

def export_as_gif(images, filename, frames_per_second=10):
    """
    Exports the supplied images as a GIF animation.
    """
    images += images[2:-1][::-1]
    images[0].save(filename, save_all=True, append_images=images[1:],
        duration=1000 // frames_per_second, loop=0)

# Export the animation as a GIF and display it on the screen.
export_as_gif(animation, ANIMATION_FILENAME, frames_per_second=FRAMES_PER_SECOND)
```

```
25/25 [==============================] - 42s 2s/step
25/25 [==============================] - 43s 2s/step
25/25 [==============================] - 43s 2s/step
25/25 [==============================] - 43s 2s/step
25/25 [==============================] - 43s 2s/step
25/25 [==============================] - 44s 2s/step
25/25 [==============================] - 44s 2s/step
25/25 [==============================] - 44s 2s/step
25/25 [==============================] - 44s 2s/step
25/25 [==============================] - 44s 2s/step
25/25 [==============================] - 44s 2s/step
25/25 [==============================] - 44s 2s/step
25/25 [==============================] - 44s 2s/step
25/25 [==============================] - 44s 2s/step
```

Figure: Latent space walking from a Ford truck to an electric car.

7.22.8 References

1. High-performance image generation using Stable Diffusion in KerasCV, available at https://keras.io/guides/keras_cv/generate_images_with_stable_diffusion/.
2. Diffusion Models Made Easy, J. Rafid Siddiqui, available at https://towardsdatascience.com/diffusion-models-made-easy-8414298ce4da.
3. How Diffusion Models Work: The Math From Scratch, available at https://theaisummer.com/diffusion-models/.
4. The Illustrated Stable Diffusion, Jay Alammar, available at https://jalammar.github.io/illustrated-stable-diffusion/.
5. What are Stable Diffusion Models and Why are they a Step Forward for Image Generation?, J. Rafid Siddiqui, available at https://towardsdatascience.com/

what-are-stable-diffusion-models-and-why-are-they-a-step-forward-for-image-generation-aa1182801d46.

6. Generating Images using Stable Diffusion, available at https://github.com/svpino/stable-diffusion/blob/main/stable-diffusion.ipynb.

BACK TO TOP

7.23 Lecture 23 - Large Language Models

7.23.1 23.1 Introduction to LLMs

Large Language Models (LLMs) are a class of Deep Neural Networks designed to understand and generate natural human language. These models have achieved state-of-the-art performance across various NLP tasks.

LLMs are a result of many years of research and advancement in NLP and Machine Learning. Important phases in the development include:

- *Statistical language models (1980s-2000s)*: developed to predict the probability of a word in a text sequence based on the preceding words. Examples of statistical language models include bag-of-words models based on N-grams. These models were used in tasks like speech recognition and machine translation, but struggled with capturing long-range dependencies and context-related information in text.
- *Neural network models (2000-2017)*: fully-connected NNs and recurrent NNs emerged as an alternative to statistical language models. Long Short-Term Memory (LSTM) RNN models were used for sequence-to-sequence tasks like machine translation and they formed the basis for several early LLMs. Similar to statistical language models, RNNs struggled with capturing context-related information, and other limitations of RNNs include the inability to parallelize the data processing, and the gradients can become unstable during training.

- *Transformer network models (2017-present)*: Transformer architecture introduced the self-attention mechanism as a replacement for the recurrent layers in RNNs. This breakthrough enabled the development of more powerful and efficient LLMs, laying the foundation for BERT, GPT, and modern LLMs.

23.1.1 Architecture of Large Language Models

The architecture of LLMs is based on Transformer Networks, which we covered in Lecture 20. The main components of the Transformer Networks architecture include:

- **Input embeddings**, are fixed-size continuous vector embeddings that represent tokens in input text.
- **Positional encodings**, are fixed-size continuous vectors that are added to the input embeddings to provide information about the relative positions of the tokens in the input text sequence.
- **Encoder**, is composed of a stack of multi-head attention modules and fully-connected (feed-forward) modules. The encoder block also includes dropout layers, residual connections, and applies layer normalization.
- **Decoder**, is composed of a stack of multi-head self-attention modules and fully-connected (feed-forward) modules similarly to the encoder block. The decoder block has an additional masked multi-head attention module, that applies masking to the next words in the text sequence to ensure that the module does not have access to those words for predicting the next token.
- **Output fully-connected layer**, the output of the decoder is passed through a fully-connected (dense, linear) layer to produce the next token in the text sequence.

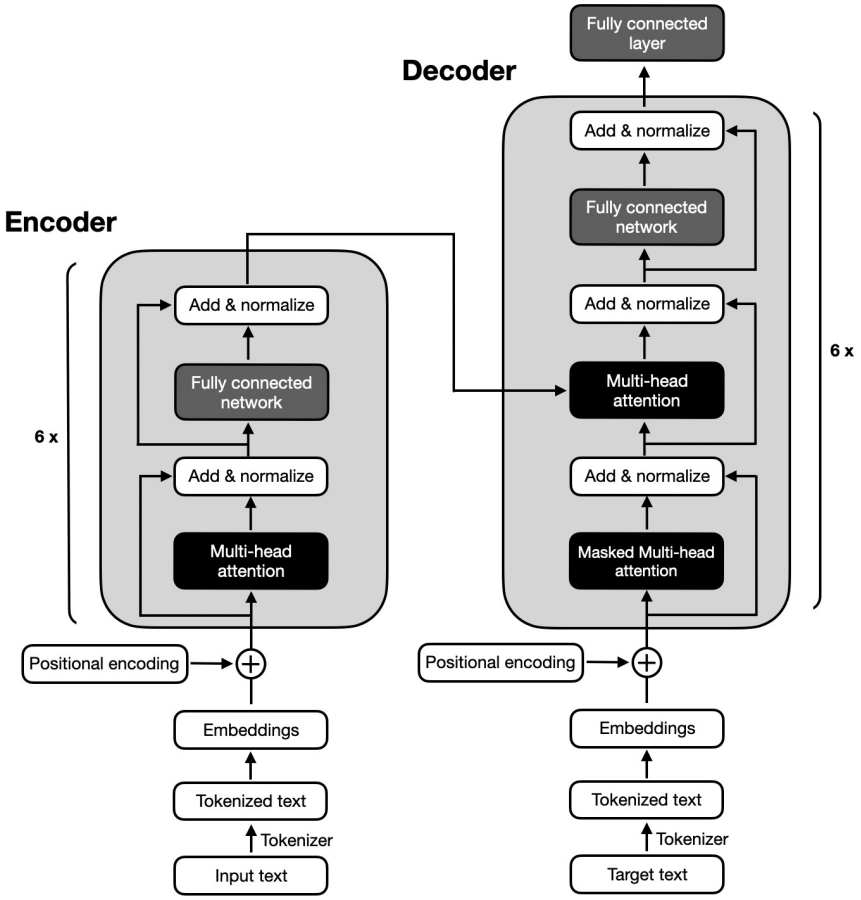


Figure: Pretraining LLMs. Source: [2].

The architecture of Transformer Networks includes multiple successive encoder and decoder blocks to create deep networks with many layers that allow learning complex patterns in input text. For example, the original Transformer Network has 6 encoder and 6 decoder blocks, as shown in the above figure.

The **self-attention mechanism** is a key component of the Transformer Network architecture that enables the model to weigh the importance of each token with respect to the other tokens in a sequence. It allows to capture long-range dependencies and relationships between the tokens (words) and helps the model to understand the context and structure of the input text sequence.

23.1.2 Variants of Transformer Network Architectures

Various LLMs have been built on top of the Transformer Network architecture. The popular variants include:

- **Decoder-only models**: are autoregressive models that utilize only the decoder part of the Transformer Network architecture. These models are particularly suitable for generating text and content. An example of decoder-only LLMs is the family of GPT models.
- **Encoder-only models**: use only the encoder part of the Transformer Network architecture, and perform well on tasks related to language understanding, such as classification and sentiment analysis. An example is the BERT model.
- **Encoder-decoder models**: employ the original Transformer Network architecture and combine encoder and decoder sub-networks, enabling to both understand language and generate content. These models can be used for various NLP tasks with minimal task-specific modifications. An example of this class of models is T5 (Text-to-Text Transfer Transformer).

List of LLMs

A large number of LLMs were developed in the past several years. Some of the most well-known LLMs include:

- *GPT* (Generative Pretrained Transformers): Developed by OpenAI, the GPT family are the best-known LLMs. They include GPT 1, 2, 3, 3.5 (initial ChatGPT), and 4 (current ChatGPT). According to some sources, GPT-4 has 1.76 trillion parameters, and it is trained on 13T tokens.
- *BERT* (Bidirectional Encoder Representations from Transformers): Developed by Google in 2018, BERT is an early LLM with 340M parameters that can understand natural language and answer questions.
- *LlaMA (Large Language Model Meta AI)*: Developed by Meta AI, LlaMA is an open-source LLM, which can be used for both research and commercial uses. It consists of several models including LlaMA base model, LlaMA-Chat, and Code-LlaMA. LlaMA 2 includes models with 7B, 13B, and 70B parameters, trained on 2T tokens.
- *Falcon*: Developed by UAE's Technology Innovation Institute (TII), it is an open-source family of models with 1.3B, 7.5B, 40B, and 180B parameters, trained on 3.5T tokens.
- *Bard*, developed by Google, Bard is an LLM with 137B parameters trained on 1.56T tokens, that is based on the LaMDA model.
- *Claude*: developed by Anthropic AI, Claude is an LLM with 137B parameters.
- *PaLM (Pathways Language Model)*: Developed by Google, PaLM is an LLM with 540B parameters capable of common-sense and arithmetic reasoning, code generation, and translation. It was trained on 3.6T tokens.
- *Cohere LLM*: Developed by Cohere, it is a family of LLMs with 6B, 13B, and 52B parameters, designed for enterprise use cases.
- *Vicuna*: Developed by LMSYS, Vicuna is a 13B parameters chat assistant finetuned from LLaMA on user-shared conversations.
- *Alpaca*: Developed by Stanford, it is an LLM finetuned from instruction-following samples by LLaMA.

- *Dolly*: Developed by Databricks, it is an open-source instruction-following LLM language model with 12B parameters.

7.23.2 23.2 Creating LLMs

Creating modern LLMs such as ChatGPT or LlaMA 2, typically involves three main phases:

1. **Pretraining**, the model extracts knowledge from large unlabeled text datasets.
2. **Supervised finetuning**, the model is refined to improve the quality of generated responses.
3. **Alignment**, the model is further refined to generate safe and helpful responses that are aligned with human preferences.

23.2.1 Pretraining

The first step in creating LLMs is **pretraining** the model on massive amounts of text data. The datasets usually consist of a large collection of web pages or e-books comprising billions or trillions of tokens, and ranging from gigabytes to terabytes of text. During pretraining, the model learns the structure of the language, grammar rules, facts about the world, and reasoning abilities. And, it also learns biases and harmful content present in the training data.

Pretraining is performed using unsupervised learning techniques. Two common approaches for pretraining LLMs are:

- **Causal Language Modeling**, also known as autoregressive language modeling, involves training the model to predict the next token in the text sequence given the previous tokens. This approach is used for pretraining ChatGPT, LlaMA 2, and it is more common with modern LLMs.
- **Masked Language Modeling**, where a certain percentage of the input tokens are randomly masked, and the model is trained to predict the masked tokens based on the surrounding context. BERT and earlier LLMs were pretrained with masked language modeling.

The following figure depicts the pretraining phase with Causal Language Modeling, where the model learns to predict the next word in a sentence given the previous words.

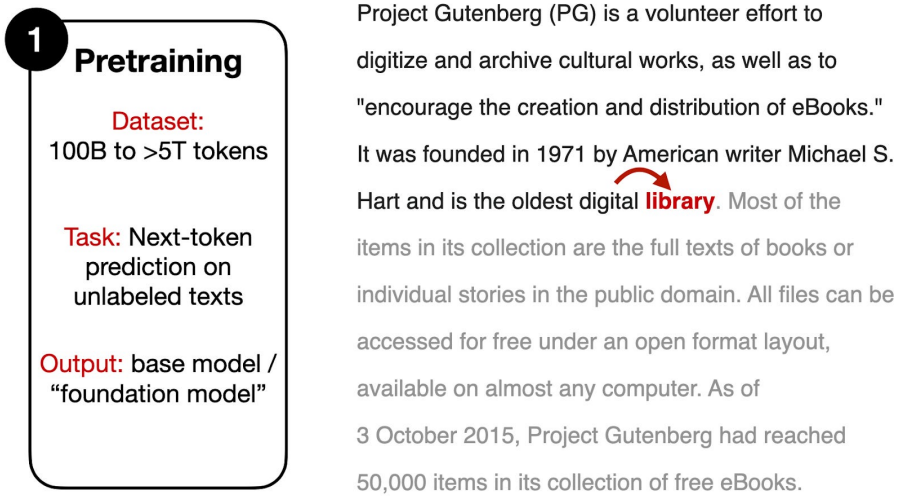


Figure: Pretraining LLMs. Source: [3].

Pretraining allows to extract knowledge from very large unlabeled datasets in unsupervised learning manner, without the need for manual labeling. Or, to be more precise, the "label" in LLMs pretraining is the next word in the text, to

which we already have access since it is part of the training text. Such pretraining approach is also called self-supervised training, since the model uses each next word in the text to self-supervise the training.

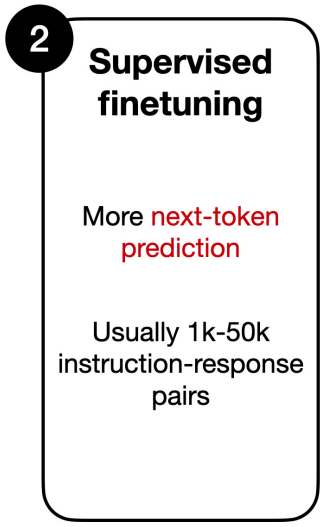
Note that pretraining LLMs from scratch is computationally expensive and time-consuming. As we stated before, the pretraining phase can cost millions of dollars (e.g., the estimated cost for training GPT-4 is $100 million). Also, pretraining LLMs requires access to large datasets and technical expertise with strong understanding of deep learning workflows, working with distributed software and hardware, and managing model training with thousands of GPUs simultaneously.

23.2.2 Supervised Finetuning

After the pretraining phase, the model is finetuned on a much smaller dataset, which is carefully generated with human supervision. This dataset consists of samples where AI trainers provide both queries (instructions) and model responses (outputs), as depicted in the following figure. That is, *instruction* is the input text given to the model, and *output* is the desired response by the model. The model takes the instruction text as input (e.g., "Write a limerick about a pelican") and uses next-token prediction to generate the output text (e.g., "There once was a pelican so fine ...").

The finetuning process involves updating the model's weights using supervised learning techniques. The objective of supervised finetuning is to improve the quality of the generated responses by the pretrained LLM.

To compile datasets for supervised finetuning, AI trainers need to write the desired instructions and responses, which is a laborious process. Typical datasets include between 1K and 100K instruction-output pairs. Based on the provided instruction-output pairs, the model is finetuned to generate responses that are similar to those provided by AI trainers.




```
{
    "instruction": "Write a limerick about a
                    pelican.",
    "input": "",
    "output": "There once was a pelican so fine,
              \nHis beak was as colorful as
              sunshine,\nHe would fish all day,\nIn
              a very unique way,\nThis pelican was
              truly divine!\n\n\n"
},


{
    "instruction": "Identify the odd one out from
                    the group.",
    "input": "Carrot, Apple, Banana, Grape",
    "output": "Carrot\n\n"
},
```

Figure: Finetuning a pretrained LLM. Source: [3].

23.2.3 Alignment with Reinforcement Learning from Human Feedback (RLHF)

To further improve the performance and align the model responses with human preferences, LLMs are typically refined in one additional phase with **Reinforcement Learning from Human Feedback (RLHF)**. This process is depicted in the figure below and involves the following steps:

1. *Collect human feedback*. For this step a new dataset is created by collecting sample prompts from a database or by creating a set of new prompts. For each prompt, multiple responses are generated by the supervised finetuned model. Next, AI trainers are asked to rank by quality all responses generated by the model for the same prompt, from best to worst. Such feedback is used to define the human preferences and expectations about the responses

by the model. Although this ranking process is time-consuming, it is usually less labor-intensive than creating the dataset for supervised finetuning, since ranking the responses is faster than writing the responses.

2. *Create a reward model*. The collected data with human feedback containing the prompts and the ranking scores of the different responses is used to train a Reward Model (denoted with RM in the figure). The task for the Reward Model is to predict the quality of the different responses to a given prompt and output a ranking score. The ranking scores provided by AI trainers are used to establish the ground-truth for training the Reward Model. Note that the Reward Model is a different model than the LLM that is being finetuned, and it only needs to rank the generated responses by the LLM.
3. *Finetune the LLM with RL*. The LLM is finetuned using a Reinforcement Learning (RL) algorithm, and for this step typically the Proximal Policy Optimization (PPO) algorithm is used. For a new prompt, the original LLM generates a response, which the Reward Model evaluates and calculates a reward score r_k. Next, the PPO algorithm uses the reward score r_k to finetune the LLM so that the total rewards for the generated responses by the LLM are maximized. I.e., the goal is to generate responses by the LLM that maximize the predicted reward scores, and by that, the responses become more aligned with human preferences and are more useful to human users.
4. *Iterative improvement*. The RLHF process is performed iteratively, with multiple rounds of collecting additional feedback from human labelers, re-training the Reward Model, and applying Reinforcement Learning. This leads to continuous refinement and improvement of the LLM's performance.

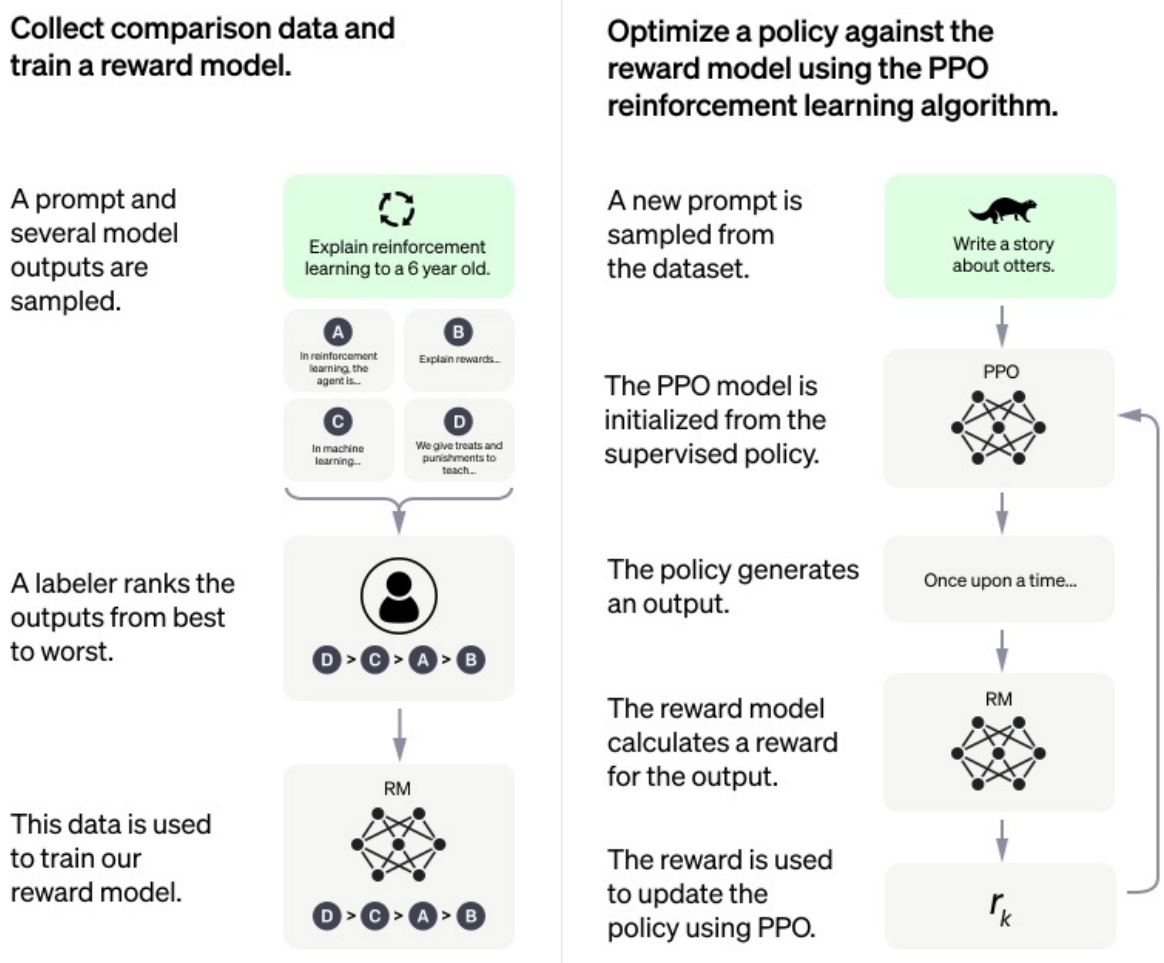


Figure: Reinforcement Learning from Human Feedback. Source: [4].

In summary, the RLHF approach creates a reward system that is augmented by human feedback and is used to teach

LLMs which responses are more aligned with human preferences. Through these iterations, LLMs can be better aligned with our human values and can lead to higher-quality responses, as well as improved performance on specific tasks.

RLHF has been successfully applied to finetune models like ChatGPT and LlaMA models. Note also that there are several variants of the RLFH approach for finetuning LLMs. For example, LlaMA 2 employs two reward models: one based on the ranks of helpfulness of the responses, and another based on the ranks of safety of the responses. The final reward score is obtained as a combination of the helpfulness and safety scores.

7.23.3 23.3 Finetuning LLMs

Finetuning LLMs involves updating the weights of an LLM model on new data to improve its performance on a specific task and make the model more suitable for a specific use case. It involves additional re-training of the model on a new dataset that is specific to that task. That is, finetuning is a transfer learning technique, where the gained knowledge by a trained model is transfered to improve the performance on a target task.

To adapt LLMs to a custom task, different finetuning techniques have been applied. *Full model finetuning* is a method that finetunes all the parameters of all the layers of a pretrained model. Full model finetuning typically can achieve the best performance, but it is also the most resource-intensive and time-consuming. *Performance-efficient finetuning* involves updating only a small number of the parameters to reduce the required computational resources and costs.

In this section, we will demonstrate how to finetune **LlaMA 2** (Large Language Model Meta AI 2), which is an open-source LLM developed by Meta AI. Released in July 2023, LlaMA 2 was the first LLM that is open for both research and commercial use. LlaMA 2 is a successor model to the original LlaMA developed by Meta AI as well. LlaMA 2 has three variants with 7B, 13B, and 70B parameters. It has been trained on 2 trillion tokens, and it has a context window of 4,096 tokens enabling to process large documents. For instance, for the task of summarizing a pdf document the context can include the entire text of the pdf document, or for dialog with a chatbot the context can include the previous conversation history with the chatbot.

Furthermore, several specialized versions of LlaMA 2 were recently released, including LlaMA-2-Chat optimized for dialog generation, and Code LlaMA optimized for code generation tasks.

23.3.1 Parameter-Efficient Finetuning (PEFT)

Finetuning LLMs is challenging since the large number of parameters of modern LLMs requires substantial computational resources for storing the models and for re-training the wights. Thus, it can be prohibitively expensive for most users. For instance, to load the largest version of the LLaMA 2 model with 70 billion parameters into the GPU memory requires approximately 280 GB of RAM. Full model finetuning of LlaMA 2 model with 70 billion parameters requires 780 GB of GPU memory. This is equivalent to 10 A100s GPUs that have 80 GB RAM each, or 48 T4 GPUs that have 16 GB RAM each. The free version of Google Colab offers one T4 GPU with 16 GB RAM.

Fortunately, several Parameter-Efficient FineTuning (PEFT) techniques have been introduced recently, which allow updating only a small number of the model weights. Consequently, these techniques enable finetuning LLMs using lower computational resources by reducing memory usage and speeding up the training process. PEFT techniques include prompt tuning, prefix tuning, adding additional adapter layers in the transformer block, and low-rank adaptation (LoRA). Among these techniques, LoRA finetuning has been the most popular, since it allows to train LLMs with a single GPU.

Hugging Face has developed a PEFT library that contains implementations of common finetuning techniques, such as weight quantization, LoRA, QLoRA, prefix tuning, and others. We will use the PEFT library to finetune LlaMA 2 on a custom dataset.

23.3.2 Low-Rank Adaptation (LoRA)

Low-Rank Adaptation (LoRA) involves freezing the pretrained model and finetuning a small number of additional weights. After the additional weights are updated, these weights are merged with the weights of the original model.

This is depicted in the following figure, where regular finetuning is shown in the left figure, and it involves updating all weights W in a pretrained model. As we know, the weight update matrix ∇W is calculated based on the negative gradient of the loss function. Finetuning with LoRA is shown in the right figure, where the weight update matrix ∇W is decomposed into two smaller matrices, $\nabla W = W_A * W_B$, with size $W_A \in \mathbb{R}^{A \times r}$ and $W_B \in \mathbb{R}^{r \times B}$. The matrices W_A and W_B are called low-rank adapters, since they have lower rank r in comparison to the original weight matrix, i.e., they have fewer number of columns or rows, respectively. During training, gradients are backpropagated only through the matrices W_A and W_B, while the pretrained weights W remain frozen.

For instance, if the full weight matrix W is of size 100×100, this is equal to $10,000$ elements (model weights). If we decompose the weight update matrix ∇W by using rank $r = 5$, the total number of elements of $W_A \in \mathbb{R}^{100 \times 5}$ and $W_B \in \mathbb{R}^{5 \times 100}$ will be $500 + 500 = 1,000$. Hence, with LoRA the number of elements was reduced from $10,000$ to $1,000$.

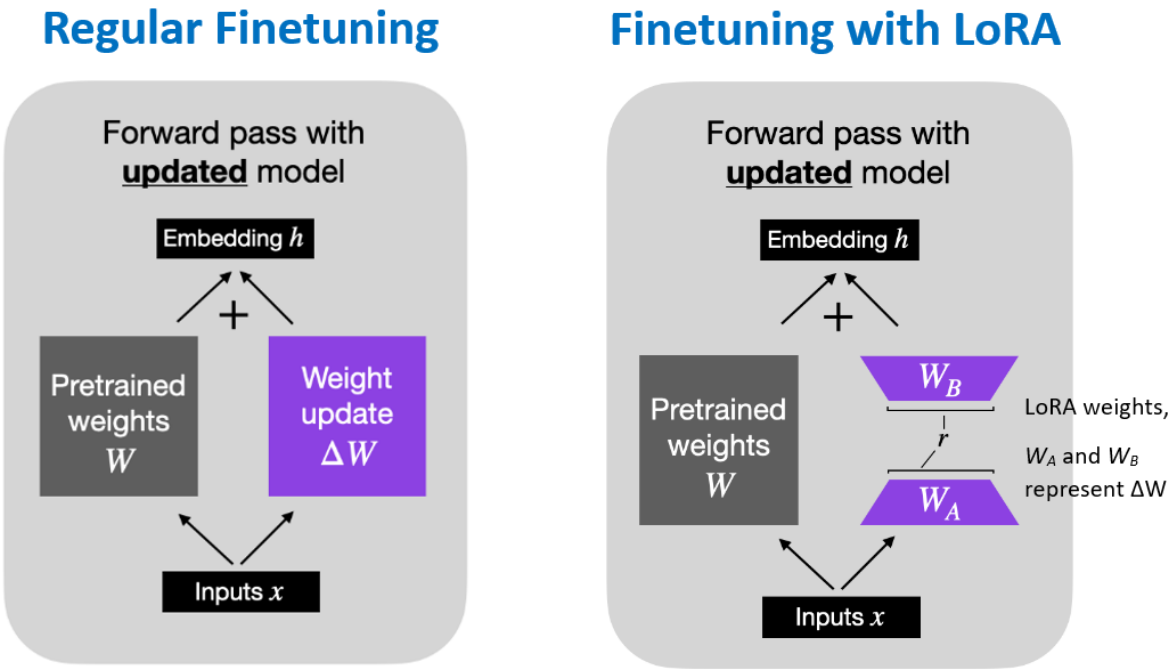


Figure: Regular finetuning versus LoRA finetuning . Source: [5].

23.3.3 Quanitized LoRA (QLoRA)

Quanitized LoRA (QLoRA) is a modified version of LoRA that uses 4-bit quantized weights. *Quantization* reduces the precision for the values of the network weights. In TensorFlow and PyTorch, the network weights by default are stored with 32-bit floating-point precision. With quantization techniques, the network weights are stored with lower precision, such as 16-bit, 8-bit, or 4-bit precision.

QLoRA introduces a new 4-bit quantization format called "nf4" (normalized float 4) where the range of values is normalized to the range [-1, 1] by dividing the values evenly into 16 bins (4-bit allows $2^4 = 16$ values). While 4-bit floating point precision (fp4) applies non-linear floating point representation of the original values and results in unequal spacing of the values, normalized float 4 precision (nf4) applies linear quantization of the original values into equaly spaced bins and follows a normal distribution.

In addition, QLoRA combines 4-bit quantization of the model weights in the pretrained model and LoRA that adds low-rank adaptor layers. The benefits of QLoRA with 4-bit quantization of the model weights include reduced size of the model and increased inference speed, while having a modest decrease in the overall model performance.

For example, with QLoRA a 70B parameter model can be finetuned with 48 GB VRAM, in comparison to 780 GB VRAM required for finetuning all weights of the original model (using 32-bit floating-point precision). Similarly, QLoRA enables to train the smaller version of LlaMA 2 with 7B parameters on a T4 GPU (provided by Google Colab) that has 16 GB VRAM. In cases when only a single GPU is available, using quantization is necessary for finetuning LLMs.

23.3.4 Finetuning Example: Finetuning LlaMA-2 7B

Import Libraries

We will begin by installing the required libraries and importing modules from these packages. These include `accelerate` (for optimized training on GPUs), `peft` (for Parameter-Efficient Fine-Tuning), `bitsandbytes` (to quantize the LlaMA model to 4-bit precision), `transformers` (for working with Transformer Networks), and `trl` (for supervised finetuning, where trl stands for Transformer Reinforcement Learning).

[ ]:
```
!pip install -q accelerate==0.21.0 peft==0.4.0 bitsandbytes==0.40.2 transformers==4.31.0␣
↪trl==0.4.7
```

```
     244.2/244.2 kB 4.8 MB/s eta 0:00:00
     72.9/72.9 kB 11.3 MB/s eta 0:00:00
     92.5/92.5 MB 19.7 MB/s eta 0:00:00
     7.4/7.4 MB 106.8 MB/s eta 0:00:00
     77.4/77.4 kB 10.4 MB/s eta 0:00:00
     1.3/1.3 MB 81.1 MB/s eta 0:00:00
     311.7/311.7 kB 37.3 MB/s eta 0:00:00
     7.8/7.8 MB 108.6 MB/s eta 0:00:00
     521.2/521.2 kB 60.7 MB/s eta 0:00:00
     115.3/115.3 kB 17.2 MB/s eta 0:00:00
     134.8/134.8 kB 20.7 MB/s eta 0:00:00
```

[ ]:
```
import os
import torch
from datasets import load_dataset
from transformers import (AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig,
    HfArgumentParser, TrainingArguments, pipeline, logging)
from peft import LoraConfig, PeftModel, get_peft_model
from trl import SFTTrainer
```

Load the Model

We will download the smallest version of LlaMA-2-Chat model with 7B parameters from Hugging Face. Understandably, the larger LlaMA 2 models with 13B and 70B parameters require larger memory and computational resources for finetuning.

Also, we will use the BitsAndBytes library to apply quantization with 4-bit precision format for loading the model weights. Loading a quantized model reduces the GPU memory requirement and makes it possible to train the model with a single GPU, as a tradeoff for some loss in precision. In the next cell we define the configuration for BitsAndBytes,

and afterward we will use the configuration in the `from_pretrained` function to load the LlaMA 2 model. The parameters in BitsAndBytes configuration are described in the commented code below.

The compute type can be either "float16", "bfloat16", or "float32" because computations are performed in either 16 or 32-bit precision. In this case, we specified to use `"torch. float16"` compute data type (i.e., 16-bit floating-point numbers) for memory-saving purposes. Note that although the model weights are loaded with 4-bit precision, the weights are dequantized to 16-bit precision for performing the calculations for the forward and backward passes through the network, as 4-bit precision is too low for performing the calculations.

[ ]:
```
# The model is Llama 2 from the Hugging Face hub
model_name = "NousResearch/Llama-2-7b-chat-hf"
```

[ ]:
```
# BitsAndBytes configuraton
bnb_config = BitsAndBytesConfig(
    # Load the model using 4-bit precision
    load_in_4bit=True,
    # Quantization type (fp4 or nf4)
    # nf4 is "normalized float 4" format, uses a symmetric quantization scheme with 4-
↪bit precision
    bnb_4bit_quant_type="nf4",
    # Compute dtype for 4-bit models
    bnb_4bit_compute_dtype= torch.float16,
    # Use double quantization for 4-bit models
    # Double quantization applies further quantization to the quantization constants
    bnb_4bit_use_double_quant=False,
)
```

We will use `AutoModelForCausalLM` to load the model with the `from_pretrained` function, and we will use the above BitesAndBytes configuration to load the model parameters with 4-bit precision.

In the following cell we will load the corresponding tokenizer for LlaMA 2 by using `AutoTokenizer` and `from_pretrained`.

[ ]:
```
# Load Llama 2 model from Hugging Face
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    # Apply quantization by using the bnb configuration from the previous cell
    quantization_config=bnb_config,
    # Don't cache the model weights, load the model weights from Hugging Face
    use_cache=False,
    # Trade-off parameter in Llama-2, less important, it should be 1 in most cases
    pretraining_tp=1,
    # Load the entire model on the GPU if available
    device_map="auto"
)
```

```
(...)ma-2-7b-chat-hf/resolve/main/config.json:   0%|          | 0.00/583 [00:00<?, ?B/s]
(...)esolve/main/model.safetensors.index.json:   0%|          | 0.00/26.8k [00:00<?, ?B/
↪s]
Downloading shards:   0%|          | 0/2 [00:00<?, ?it/s]
model-00001-of-00002.safetensors:   0%|          | 0.00/9.98G [00:00<?, ?B/s]
model-00002-of-00002.safetensors:   0%|          | 0.00/3.50G [00:00<?, ?B/s]
```

```
Loading checkpoint shards:   0%|          | 0/2 [00:00<?, ?it/s]
(...)t-hf/resolve/main/generation_config.json:   0%|          | 0.00/179 [00:00<?, ?B/s]
```

[ ]:
```
# Load tokenizer from Hugging Face
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
# Needed for LLaMA tokenizer
tokenizer.pad_token = tokenizer.eos_token
# Fix an overflow issue with fp16 training
tokenizer.padding_side = "right"
```

```
(...)at-hf/resolve/main/tokenizer_config.json:   0%|          | 0.00/746 [00:00<?, ?B/s]
tokenizer.model:   0%|          | 0.00/500k [00:00<?, ?B/s]
(...)2-7b-chat-hf/resolve/main/tokenizer.json:   0%|          | 0.00/1.84M [00:00<?, ?B/
↪s]
(...)b-chat-hf/resolve/main/added_tokens.json:   0%|          | 0.00/21.0 [00:00<?, ?B/s]
(...)-hf/resolve/main/special_tokens_map.json:   0%|          | 0.00/435 [00:00<?, ?B/s]
```

Define LoRA Configuration

Next, the model will be packed into the LoRA format, which will introduce additional weights and keep the original weights frozen. The parameters in the LoRA configuration include:

- `r`, determines the rank of update matrices, where lower rank results in smaller update matrices with fewer trainable parameters, and greater rank results in more trainable parameters but more robust model.
- `lora_alpha`, controls the LoRA scaling factor.
- `lora_dropout`, is the dropout rate for LoRA layers.
- `bias`, specifies if the bias parameters should be trained.
- `task_type`, is Causal LLM for the considered task.

[ ]:
```
# LoRA configuration
peft_config = LoraConfig(
    # LoRA rank dimension
    r=64,
    # Alpha parameter for LoRA scaling
    lora_alpha=16,
    # Dropout rate for LoRA layers
    lora_dropout=0.1,
    bias="none",
    task_type="CAUSAL_LM",
)
```

In order to understand how LoRA impacts the finetuning of LlaMA 2 model, let's compare the total number of trainable parameters in LLaMA 2 and the trainable parameters for the LoRA model. As we can note in the cell below, the LoRA model has about 33M trainable parameters, which is about 1% of the total trainable parameters in LlaMA 2. This makes it possible to finetune the model on a single GPU.

[ ]:
```
def print_number_of_trainable_model_parameters(model, use_4bit=True):
    trainable_model_params = 0
    all_model_params = 0
    for _, param in model.named_parameters():
        all_model_params += param.numel()
        if param.requires_grad:
            trainable_model_params += param.numel()
    if use_4bit:
        all_model_params *= 2
        trainable_model_params *= 2
    print(f"Total model parameters: {all_model_params:,d}. Trainable model parameters:
↪{trainable_model_params:,d}. Percent of trainable parameters: {100 * trainable_model_
↪params/ all_model_params:4.2f} %")
```

[ ]:
```
# compare the number of trainable parameters to QLoRA model
qlora_model = get_peft_model(model, peft_config)

# print trainable parameters
print_number_of_trainable_model_parameters(qlora_model)
```

```
Total model parameters: 7,067,934,720. Trainable model parameters: 67,108,864. Percent␣
↪of trainable parameters: 0.95 %
```

Load the Dataset

We will use the Lamini docs dataset, which contains questions and answers about the framework Lamini for training and developing Language Models. The dataset contains 1,260 question/answer pairs. Here are a few samples from the dataset.

Question	Answer
Does Lamini support generating code	Yes, Lamini supports generating code through its API.
How do I report a bug or issue with the Lamini documentation?	You can report a bug or issue with the Lamini documentation by submitting an issue on the Lamini GitHub page.
Can Lamini be used in an online learning setting, where the model is updated continuously as new data becomes available?	It is possible to use Lamini in an online learning setting where the model is updated continuously as new data becomes available. However, this would require some additional implementation and configuration to ensure that the model is updated appropriately and efficiently.

A preprocessed version of the dataset in a format that matches the instruction-output pairs for LlaMA 2 is available on Hugging Face, and we will directly load the preprocessed version of the dataset.

[ ]:
```
# Lamini dataset
dataset = load_dataset("mwitiderrick/llamini_llama", split="train")
```

```
Downloading data files:   0%|          | 0/1 [00:00<?, ?it/s]
```

```
Downloading data:   0%|          | 0.00/1.14M [00:00<?, ?B/s]
```

```
Extracting data files:   0%|          | 0/1 [00:00<?, ?it/s]
```

```
Generating train split: 0 examples [00:00, ? examples/s]
```

[ ]:
```
print(f'Number of prompts: {len(dataset)}')
```

```
Number of prompts: 1260
```

Model Training

The next cell defines the training arguments, and the commented notes describe the arguments. Note that we will finetune the model for only 1 epoch (if we finetune for more than 1 epoch it will take longer but it will probably result in improved performance).

[ ]:
```
# Set training parameters
training_arguments = TrainingArguments(
    # Output directory where the model predictions and checkpoints will be stored
    output_dir="./results",
    # Number of training epochs
    num_train_epochs=1,
    # Batch size per GPU for training
    per_device_train_batch_size=4,
    # Number of update steps to accumulate the gradients for
    gradient_accumulation_steps=1,
    # Optimizer to use
    optim="paged_adamw_32bit",
    # Save checkpoint every number of steps
    save_steps=0,
    # Log updates every number of steps
    logging_steps=25,
    # Initial learning rate (AdamW optimizer)
    learning_rate=2e-4,
    # Weight decay to apply
    weight_decay=0.001,
    # Enable fp16/bf16 training (set bf16 to True with an A100)
    fp16=False,
    bf16=False,
    # Maximum gradient normal (gradient clipping)
    max_grad_norm=0.3,
    # Group sequences with same length into batches (to minimize padding)
    # Saves memory and speeds up training considerably
    group_by_length=True,
    # Learning rate schedule
    lr_scheduler_type="constant"
)
```

Next, we will use the `SFTTrainer` class in Hugging Face to create an instance of the model by passing the loaded LlaMA 2 model, training dataset, PeFT configuration, tokenizer, and the training arguments. `SFTTrainer` stands for Supervised Fine-Tuning Trainer.

[ ]:
```
# Set supervised finetuning parameters
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    peft_config=peft_config,
    tokenizer=tokenizer,
```

(continues on next page)

(continued from previous page)

```
    args=training_arguments,
    # Column in the dataset that contains the data
    dataset_text_field="text",
    # Maximum sequence length to use
    max_seq_length=None,
    # Pack multiple short examples in the same input sequence to increase efficiency
    packing=False,
)
```

```
/usr/local/lib/python3.10/dist-packages/peft/utils/other.py:102: FutureWarning: prepare_
↪model_for_int8_training is deprecated and will be removed in a future version. Use␣
↪prepare_model_for_kbit_training instead.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/trl/trainer/sft_trainer.py:159: UserWarning: You␣
↪didn't pass a `max_seq_length` argument to the SFTTrainer, this will default to 1024
  warnings.warn(
```

```
Map:   0%|          | 0/1260 [00:00<?, ? examples/s]
```

Finally, we can train the model with the `train()` function in Hugging Face. In the output of the cell we can see the loss for every 25 training steps, because we set `logging_steps=25` in the training arguments.

The training took about 15 minutes on a T4 GPU with High-RAM memory on Google Clab Pro.

[ ]:
```
# Train the model
trainer.train()
```

```
You're using a LlamaTokenizerFast tokenizer. Please note that with a fast tokenizer,␣
↪using the `__call__` method is faster than using a method to encode the text followed␣
↪by a call to the `pad` method to get a padded encoding.
/usr/local/lib/python3.10/dist-packages/torch/utils/checkpoint.py:429: UserWarning:␣
↪torch.utils.checkpoint: please pass in use_reentrant=True or use_reentrant=False␣
↪explicitly. The default value of use_reentrant will be updated to be False in the␣
↪future. To maintain current behavior, pass use_reentrant=True. It is recommended that␣
↪you use use_reentrant=False. Refer to docs for more details on the differences between␣
↪the two variants.
  warnings.warn(
```

```
<IPython.core.display.HTML object>
```

```
TrainOutput(global_step=315, training_loss=0.6278291732545883, metrics={'train_runtime':␣
↪1020.6457, 'train_samples_per_second': 1.235, 'train_steps_per_second': 0.309, 'total_
↪flos': 5614695674511360.0, 'train_loss': 0.6278291732545883, 'epoch': 1.0})
```

Generate Text

To generate text with the trained model we will use the Hugging Face `pipeline` with the task set to `"text-generation"`. We can set the length of the generated text tokens with the `max_length` argument.

The output displays the start `<s>[INST]` and end `[/INST]` of the instruction prompt, followed by the generated output by the model.

[ ]:
```
# Run text generation pipeline with the finetuned model
prompt = "What are Lamini models?"
```

(continues on next page)

(continued from previous page)

```
pipe = pipeline(task="text-generation", model=model, tokenizer=tokenizer, max_length=200)
output = pipe(f"<s>[INST] {prompt} [/INST]")
print(output[0]['generated_text'])
```

```
Xformers is not installed correctly. If you want to use memory_efficient_attention to␣
↪accelerate training use the following command to install Xformers
pip install xformers.
/usr/local/lib/python3.10/dist-packages/transformers/generation/utils.py:1270:␣
↪UserWarning: You have modified the pretrained model configuration to control␣
↪generation. This is a deprecated strategy to control generation and will be removed␣
↪soon, in a future version. Please use a generation configuration file (see https://
↪huggingface.co/docs/transformers/main_classes/text_generation )
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torch/utils/checkpoint.py:61: UserWarning: None␣
↪of the inputs have requires_grad=True. Gradients will be None
  warnings.warn(
```

```
<s>[INST] What are Lamini models? [/INST]  Lamini is a language model training platform␣
↪that allows developers to train and fine-tune their own custom language models using␣
↪their own data. everybody can train a model with Lamini, regardless of their technical␣
↪expertise. Lamini models are trained on large datasets of text and can be used for a␣
↪variety of natural language processing tasks, such as text classification, sentiment␣
↪analysis, and language translation.

Lamini models are trained using a technique called "fine-tuning," which involves␣
↪adjusting the weights of a pre-trained language model to fit a specific task or␣
↪dataset. This allows developers to train a model that is tailored to their specific␣
↪needs, without having to start from scratch.

Lamini models can be trained on any dataset, including text data from the internet,␣
↪social media, or internal company data. They can also be trained on a combination of␣
↪different datasets, allowing developers
```

[ ]:
```
# Run text generation pipeline with the finetuned model
prompt = "How to evaluate the quality of the generated text with Lamini models"
pipe = pipeline(task="text-generation", model=model, tokenizer=tokenizer, max_length=500)
output = pipe(f"<s>[INST] {prompt} [/INST]")
print(output[0]['generated_text'])
```

```
<s>[INST] How to evaluate the quality of the generated text with Lamini models [/INST] ␣
↪Lamini offers a Python library called Lamini Python Library that allows you to train␣
↪and use LLMs. everybody can use Lamini's python library to train and use LLMs.

Here are some ways to evaluate the quality of the generated text with Lamini models:

1. Perplexity: Measure the perplexity of the generated text by comparing it to a␣
↪reference text. Lower perplexity indicates better quality.
2. BLEU score: Use the BLEU score to evaluate the quality of the generated text. BLEU is␣
↪a metric that measures the similarity between the generated text and a reference text.
3. ROUGE score: Use the ROUGE score to evaluate the quality of the generated text. ROUGE␣
↪is a metric that measures the similarity between the generated text and a reference␣
↪text.
4. METEOR score: Use the METEOR score to evaluate the quality of the generated text.␣
↪METEOR is a metric that measures the similarity between the generated text and a␣
↪reference text.
```

(continues on next page)

(continued from previous page)

```
5. Human evaluation: Have human evaluators rate the quality of the generated text on a␣
↪scale from 1 to 5.
6. Automatic metrics: Use automatic metrics such as perplexity, entropy, and coherence␣
↪to evaluate the quality of the generated text.
7. LLM's performance on specific tasks: Evaluate the performance of the LLM on specific␣
↪tasks such as text generation, question answering, and language translation.
8. LLM's performance on specific domains: Evaluate the performance of the LLM on␣
↪specific domains such as medical text, legal text, or technical text.
9. LLM's performance on specific types of data: Evaluate the performance of the LLM on␣
↪specific types of data such as text, images, or audio.
10. LLM's performance on specific evaluation metrics: Evaluate the performance of the␣
↪LLM on specific evaluation metrics such as accuracy, precision, recall, and F1 score.

It's important to note that the quality of the generated text can vary depending on the␣
↪specific use case and the data used to train the LLM. Therefore, it's important to␣
↪evaluate the quality of the generated text in the context
```

[ ]:
```
# Run text generation pipeline with the finetuned model
prompt = "Write a poem about Data Science"
pipe = pipeline(task="text-generation", model=model, tokenizer=tokenizer, max_length=500)
output = pipe(f"<s>[INST] {prompt} [/INST]")
print(output[0]['generated_text'])
```

```
<s>[INST] Write a poem about Data Science [/INST]  In the realm of code and algorithms,
 nobody knows like Data Science
With a dash of math and a pinch of magic,
She weaves a tale of insight and wonder.

With a click of her mouse, she uncovers
The secrets hidden deep within the data's cloak.
She sifts through the noise and finds the gems,
And turns them into insights that make us think.

She's a detective of the digital age,
Uncovering patterns and connections that we've never seen.
She's a wizard of the data realm,
Weaving spells of prediction and machine learning.

With a nod of her head, she conjures up
A world of possibilities, a world of wonder.
She's a master of the data craft,
A weaver of tales that make our hearts sing.

So let us hail the Data Queen,
Who brings us closer to the truth we seek.
For in the realm of data science,
She's the one who holds the key.
```

23.3.5 Retrieval Augmented Generation (RAG)

Retrieval Augmented Generation (RAG) refers to using external sources of information for improving the quality of generated responses by LLMs. RAG enables LLMs to retrieve facts from external sources (such as Wikipedia, news articles) and provide responses that are more accurate and/or are up-to-date.

In general, the internal knowledge of LLMs is static, as it is fixed by the date of the used training dataset. Therefore, LLMs cannot answer questions about current events, and they are stuck in the time moment of their training data. Updating LLMs with knowledge about current events requires to continuously retrain the models on new data. Such a process is very expensive, as it requires collecting updated datasets and finetuning the model to update the weights.

RAG enables to avoid expensive LLMs retraining, by retrieving information from updated external databases to generate responses. The RAG approach involves two phases: retrieval and content generation. The retrieval phase includes performing relevancy search of external databases regarding a user query, and retrieving supporting documents and snippets of important information. Afteward, in the content generation phases, these supporting documents are used as a context that is appended to the user query, and are fed to the LLMs for generating the final response.

Instead of relying only on the information contained in the training dataset used for training an LLM, RAG provides an interface to external knowledge to ensure that the model has access to the most current and reliable facts. E.g., in enterprise setting, external sources of information for RAG can comprise of various company-specific files, documents, and databases. Employing RAG can result in more relevant responses and it can reduce the problem of hallucination by LLMs. It also allows the users to review the sources that were used by LLMs and verify the accuracy of generated responses.

23.3.6 Prompt Engineering

Prompt engineering is a technique for improving the performance of LLMs by providing detailed context and information about a specific task. It involves creating text prompts that provide additional information or guidance to the model, such as the topic of the generated response. With prompt engineering, the model can better understand the kind of expected output and produce more accurate and relevant results.

The following tips for creating effective prompts as part of prompt engineering can improve the performance of LLMs:

- Use clear and concise prompts: The prompt should be easy to understand and provide enough information for the model to generate relevant output. Avoid using jargon or technical terms.
- Use specific examples: Providing specific examples can help the model better understand the expected output. For example, if you want the model to generate a story about a particular topic, include a few sentences about the setting, characters, and plot.
- Vary the prompts: Use prompts with different styles, tones, and formats to obtain more diverse outputs from the model.
- Test and refine: Test the prompts on the model and refine them by adding more detail or adjusting the tone and style.
- Use feedback: Use feedback from users or other sources to identify areas where the model needs more guidance and make adjustments accordingly.

Chain-of-thought technique involves providing the LLM with a series of instructions to help guide the model and generate a more coherent and relevant response. This technique is useful for obtaining well-reasoned responses from LLMs.

An example of a chain-of-thought prompt is as follows: "You are a virtual tour guide from 1901. You have tourists visiting Eiffel Tower. Describe Eiffel Tower to your audience. Begin with (1) why it was built, (2) how long it took to build, (3) where were the materials sourced to build, (4) number of people it took to build it, and (5) number of people visiting the Eiffel tour annually in the 1900's, the amount of time it completes a full tour, and why so many people visit it each year. Make your tour funny by including one or two funny jokes at the end of the tour."

7.23.4 23.4 Limitations and Ethical Considerations of LLMs

Although LLMs have demonstrated impressive performance across a wide range of tasks, there are several limitations and ethical considerations that raise concerns.

Limitations:

- *Computational resources*: Training LLMs requires significant computational resources, making it difficult for researchers with limited access to GPUs or specialized hardware to develop and use these models.
- *Data bias*: LLMs are trained on vast amounts of data from the internet, which often contain biases present in the data. As a result, the models may unintentionally learn and reproduce biases in their generated responses.
- *Producing hallucinations*: LLMs can produce hallucinations, which are responses that are false, inacurate, unexpected, or contextually inappropriate. One example of hallucination by ChatGPT is when asked to list academic papers by an author, and it provides papers that don't exist.
- *Inability to explain*: LLMs are inherently black-box models, making it challenging to explain their reasoning or decision-making processes, which is essential in certain applications like healthcare, finance, and legal domains.

Ethical considerations:

- *Privacy concerns*: LLMs memorize information from their training data, and can potentially reveal sensitive information or violate user privacy.
- *Misinformation and manipulation*: Text generated by LLMs can be exploited to create disinformation, fake news, or deepfake content that manipulates public opinion and undermines trust.
- *Accessibility and fairness*: The computational resources and expertise required to train LLMs may lead to an unequal distribution of benefits, where only a few organizations have the resources to develop and control these powerful models.
- *Environmental impact*: The large-scale training of LLMs consumes a significant amount of energy contributing to carbon emissions, which raises concerns about the environmental sustainability of these models.

Conclusively, it is important to encourage transparency, collaboration, and responsible AI practices to ensure that LLMs benefit all members of society without causing harm.

7.23.5 23.5 Foundation Models

Foundation Models are extremely large NN models trained on tremendous amounts of data with substantial computational resources, resulting in high capabilities for transfer learning to a wide range of downstream tasks. In other words, these models are scaled along each of the three factors: number of model parameters, size of the training dataset, and amount of computation. And, they are typically trained using self-supervised learning on unlabeled data. The scale of Foundation Models leads to new emergent capabilities, such as the ability to perform well on tasks that the models were not explicitly trained to do. This allows few-shot learning, which refers to finetuning Foundation Models to new downstream tasks by using only a few training data instances for the new task. Similarly, zero-shot learning extends this concept even further, and refers to a model's ability to generalize to new tasks for which the model hasn't seen any examples during the training.

LLMs represent early examples of Foundation Models, because LLMs are trained at scale and can be adapted for various NLP tasks, even for tasks they were not trained to perform.

The term Foundation Models is more general than LLMs, and they generally refer to large models that are trained on multimodal data, where the inputs can include text, images, audio, video, and other data sources.

The importance of Foundation Models is in their potential to replace task-specific ML models that are specialized in solving one task (i.e., optimized to perform well on one dataset) with general models that have the capabilities to solve multiple tasks. I.e., these models can serve as a foundation that is adaptable to a broad range of applications.

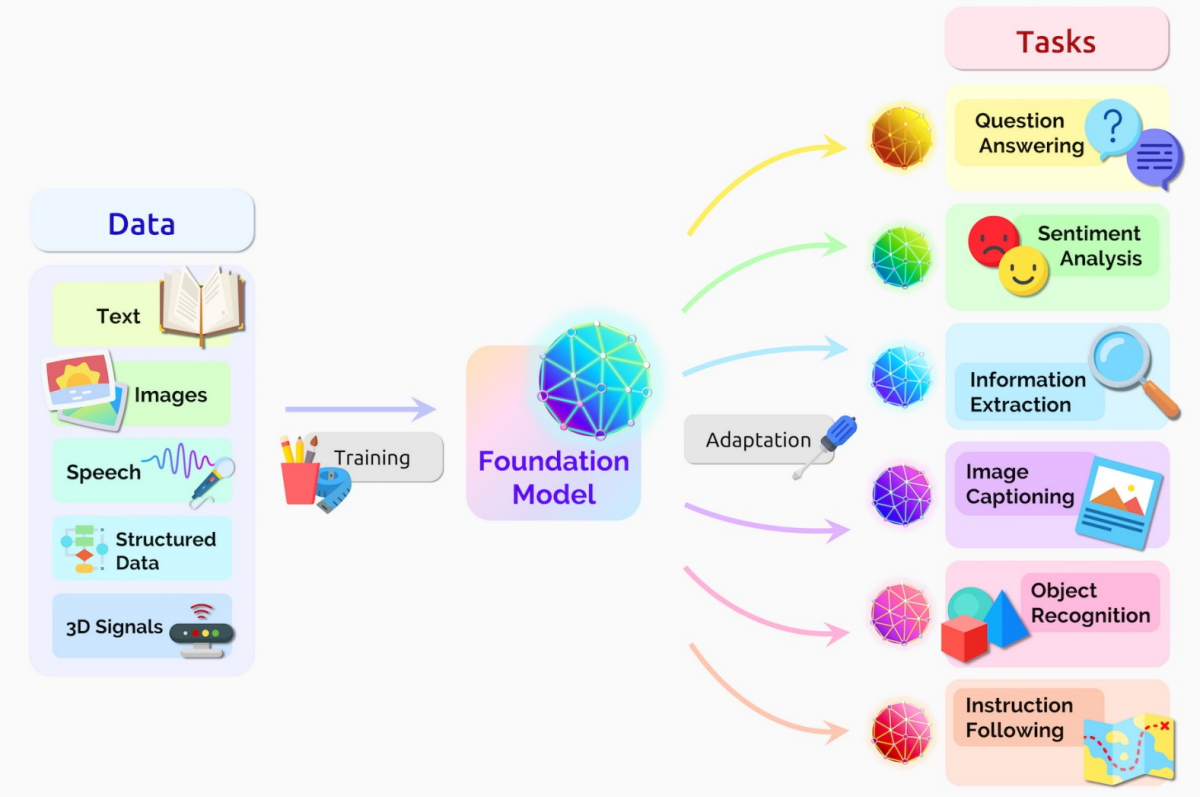


Figure: Foundation model. Source: link.

7.23.6 References

1. Introduction to Large Language Models, by Bernhard Mayrhofer, available at https://github.com/datainsightat/introduction_llm.
2. Understanding Encoder and Decoder LLMs, by Sebastian Raschka, available at https://magazine.sebastianraschka.com/p/understanding-encoder-and-decoder.
3. LLM Training: RLHF and Its Alternatives, by Sebastian Raschka, available at https://magazine.sebastianraschka.com/p/llm-training-rlhf-and-its-alternatives.
4. Training Language Models to Follow Instructions with Human Feedback, by Long Ouyang et al., available at https://arxiv.org/abs/2203.02155.
5. Parameter-Efficient LLM Finetuning With Low-Rank Adaptation (LoRA), by Sebastian Raschka, available at https://sebastianraschka.com/blog/2023/llm-finetuning-lora.html.
6. How to Fine-tune Llama 2 With LoRA, by Derrick Mwiti, available at https://www.mldive.com/p/how-to-fine-tune-llama-2-with-lora.
7. Fine-Tuning Llama 2.0 with Single GPU Magic, by Chee Kean, available at https://ai.plainenglish.io/fine-tuning-llama2-0-with-qloras-single-gpu-magic-1b6a6679d436.
8. Fine-Tuning LLaMA 2 Models using a single GPU, QLoRA and AI Notebooks, by Mathieu Busquet, available at https://blog.ovhcloud.com/fine-tuning-llama-2-models-using-a-single-gpu-qlora-and-ai-notebooks/.
9. Getting started with Llama, by Meta AI, available at https://ai.meta.com/llama/get-started/.

BACK TO TOP

7.24 Lecture 24 - Introduction to Data Science Operations (DSOps)

7.25 Lecture 25 - Deploying Projects as Web Applications

7.25.1 25.1 Introduction to Web APIs for Model Serving

After a predictive model for a Data Science (DS) project is developed, the next steps in the project life cycle include deploying the model and serving the model to the end-users. One approach to serving the model is via a web API where the end-users can submit requests to query the model, and obtain the model's predictions as responses to their queries.

In general, an **API (Application Programming Interface)** enables software applications to communicate with each other and exchange data. In the context of web APIs, the terms client and server are typically used to refer to the two main components that are involved in the communication between software applications over the web. The **client** is typically a front-end application that runs in a web browser and allows to access data or services from the server. The client sends a *request* to the server, usually in the form of an HTTP request. The **server** is the application that receives and processes requests from clients. The server performs actions based on the requests, and sends a *response* back to the client. When a response is received from the server, the client presents the data to the end-users and/or it may take other actions based on the response from the server.

Figure: Web API. Source: [4].

An example of usage of a web API in Data Science is for accessing predictions from a model that is deployed on a server. The end-user interacts with a front-end application that runs in a web browser and submits a request to initiate an API call (e.g., submit an image for classification or submit tabular data inputs for classification). The API calls the server to obtain predictions from the model. After the model makes the prediction, the API transfers the prediction to the end-user in the form of JSON, XML, CSV, or another format.

Similarly, another use of web APIs in Data Science is for accessing data from a database that is hosted on a server, and afterward using the retrieved data to train a predictive model.

25.1.1 REST Architecture

Two common approaches for building web APIs are SOAP (Service Object Access Protocol) and REST (REpresentational State Transfer). Whereas SOAP is a protocol for communication between computer systems, REST is an architecture that defines a set of constraints for communication over a network and promotes simplicity, scalability, and reliability. Because of that, REST has been very popular, since it introduces guidelines regarding the architecture of an API, instead of defining rules of specifications for the data exchange.

REST (REpresentational State Transfer) defines the following architectural constraints:

- **Stateless**: The server won't maintain any state between requests from the client, and every request must contain all necessary information. I.e., the server cannot use information from previous requests by the client.
- **Client-server design**: The client and server must be decoupled from each other. The decoupling helps them to evolve independently.
- **Cacheable**: The response from the server should be cacheable, and it can be reused later by the client, if needed.
- **Uniform interface**: The data transfer from the server to the client will be in a standardized format, instead of a format that is specific to one application.
- **Layered system**: The client may access the resources on the server indirectly through other layers. Also, the server can have multiple layers, such as security, application, business logic, and these layers must remain invisible to the client.
- **Code on demand**: The client may extend functionalities by downloading code blocks from the server.

An API that adheres to the REST guidelines is referred to as **RESTful API** or simply REST API.

RESTful APIs provide access to web services through public web URLs, referred to as request endpoints. The client sends an HTTP request to the specific URL, and the RESTful API uses HTTP methods to manage the resources in the web service. Although there are many HTTP methods, the most commonly used methods with RESTful APIs include: `GET` (retrieve resources), `POST` (create a new resource), `PUT` (update an existing resource),`PATCH` (partially update an existing resource), and `DELETE` (delete a resource).

25.1.2 RESTful APIs

RESTful APIs can be developed in different programming languages. Examples of libraries include Node.js in Javascript, Roda and Sinatra in Ruby, Spring Boot in Java, etc.

In Python, the most popular libraries for developing RESTful APIs are Flask, Fast API, and Django.

Flask is a micro framework for developing RESTful APIs, which means that it is lightweight and provides the basic components for serving predictive models as web services. Flask is the easiest to learn and use in comparison to the other frameworks, and it is known for its simplicity and flexibility. However, it also has limitations in terms of functionality, since it is designed for low-volume APIs, and does not scale well to large applications. Also, Flask is a general-purpose web framework and may require importing additional libraries for specific functionalities.

Fast API is also a lightweight framework, but it is designed specifically for building web APIs quickly and efficiently. Fast API was introduced in 2018 and it is a newer framework, but it has quickly gained popularity among developers.

Its unique characteristic is its speed, as it is the fastest Python framework. FastAPI is explicitly designed for building APIs, and offers many features for automating the building of APIs.

Django is a full-stack web framework, that has many built-in features and a specific project structure. This makes it more difficult to learn than Flask and Fast API. On the other hand, Django is a powerful framework that has been used in many large-scale projects as it offers various features for building APIs. This is considered advantageous by developers who want a more structured framework or who prefer a comprehensive solution without relying on many third-party packages.

There are also other similar Python libraries for developing RESTful APIs. As well as, numerous frameworks for developing RESTful APIs as cloud services are available by the cloud providers such as Amazon SageMaker, Azure ML, Google Cloud ML, IBM Watson Cloud, and others. These cloud frameworks provide access to pretrained models, deployment of custom models, and allow easy integration of predictive models into users' applications.

In this lecture we will demonstrate building a web API for Data Science projects with Flask, due to its ease of use.

7.25.2 25.2 Deploying a Model for Iris Flowers Classification

This section was inspired by the article Deploying Machine Learning Models using Flask.

The goal is to deploy a simple Machine Learning model for classification of the Iris dataset, which we used in Lecture 13. Recall that the dataset consists of measurements of three different species of irises:

1. Iris Setosa
2. Iris Versicolour
3. Iris Virginica

Each data point has 4 features, which include:

1. Sepal length in cm
2. Sepal width in cm
3. Petal length in cm
4. Petal width in cm

Let's load the dataset using Pandas, and display the dataframe.

[1]:
```
# Importing libraries
import pandas as pd
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import LabelEncoder

import warnings
warnings.filterwarnings('ignore')

# Load the dataset
data = pd.read_csv('Iris-API/iris.csv')
```

[2]:
```
data
```

[2]:
```
     sepal.length  sepal.width  petal.length  petal.width    variety
0             5.1          3.5           1.4          0.2     Setosa
1             4.9          3.0           1.4          0.2     Setosa
2             4.7          3.2           1.3          0.2     Setosa
```

(continues on next page)

(continued from previous page)

```
3             4.6          3.1           1.5          0.2     Setosa
4             5.0          3.6           1.4          0.2     Setosa
..            ...          ...           ...          ...        ...
145           6.7          3.0           5.2          2.3  Virginica
146           6.3          2.5           5.0          1.9  Virginica
147           6.5          3.0           5.2          2.0  Virginica
148           6.2          3.4           5.4          2.3  Virginica
149           5.9          3.0           5.1          1.8  Virginica

[150 rows x 5 columns]
```

In the dataframe, the `variety` column contains the target labels. First, we will convert the labels from categorical values to numerical values by applying the scikit-learn `LabelEncoder`.

[3]:
```
# Convert the label column into ordinal format
label_feature = data[['variety']]
label_encoder = LabelEncoder()
label_encoded = label_encoder.fit_transform(label_feature)
data['variety'] = pd.DataFrame(label_encoded, columns=label_feature.columns, index=label_
↪feature.index)

data
```

[3]:
```
     sepal.length  sepal.width  petal.length  petal.width  variety
0             5.1          3.5           1.4          0.2        0
1             4.9          3.0           1.4          0.2        0
2             4.7          3.2           1.3          0.2        0
3             4.6          3.1           1.5          0.2        0
4             5.0          3.6           1.4          0.2        0
..            ...          ...           ...          ...      ...
145           6.7          3.0           5.2          2.3        2
146           6.3          2.5           5.0          1.9        2
147           6.5          3.0           5.2          2.0        2
148           6.2          3.4           5.4          2.3        2
149           5.9          3.0           5.1          1.8        2

[150 rows x 5 columns]
```

25.2.1 Model Training and Prediction

Let's extract the training features from the dataframe into variable `X` and the training labels into variable `y`.

In the following cell, we use a Logistic Regression model from scikit-learn to train a classification model.

[4]:
```
# Extract the features X and target y variables
X = data.drop(['variety'], axis=1)
y = data['variety']
```

[5]:
```
# Train a Logistic Regression model
logreg_model = LogisticRegression()
logreg_model.fit(X, y)
```

[5]:
```
LogisticRegression()
```

Finally, let's define a function `classify` that takes 4 input arguments, and predicts the class of the iris flowers. The four inputs `a`, `b`, `c`, and `d` correspond to the sepal length and width, and petal length and width. The 'classify' function first converts the inputs to a numpy array, and reshapes the array to represent a batch with 1 data instance. Afterward, the trained Logistic Regression classifier is used to predict the variety of an iris flower.

[6]:
```
# Predict the iris class based on inputs a, b, c, d
def classify(a, b, c, d):
    arr = np.array([a, b, c, d], dtype=np.float64) # Convert to numpy array
    query = arr.reshape(1, -1) # Reshape the array
    prediction = logreg_model.predict(query)[0] # Make a prediction
    variety = variety_mappings[prediction] # Map the prediction to iris variety
    return variety # Return the iris variety

# Dictionary containing the mapping
variety_mappings = {0: 'Setosa', 1: 'Versicolor', 2: 'Virginica'}
```

To make sure that the function `classify` works, in the following cell a set of four input values is passed to `classify`. We can see that the output is the iris target label, as expected.

[7]:
```
classify(1.0, 1.0, 2.0, 2.0)
```

[7]:
```
'Setosa'
```

The above code for classifying iris flowers is saved as `model.py` under the directory `IRIS-API/model.py`.

25.2.2 Creating an HTML Webpage

Next, we will build the following simple HTML page to accept inputs from the end-users.

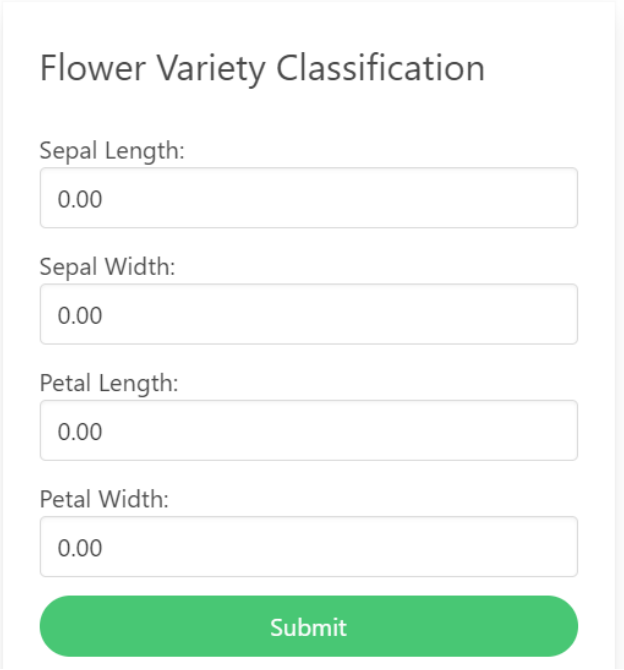


Figure: Home webpage to accept inputs from end-users.

The HTML code for the input form is shown below, containing head and body sections. The `head` defines metadata and styles with the tags `<meta>` and `<style>`.

The `body` contains `<div>` tags with `class='field'` for each of the 4 inputs: sepal/petal lengths/widths. Each input tag has `class="input"`, and `type="number"` to accept numerical values from the end-users. All the `<div>` tags are enclosed within a form with `action="predict"` and `method="GET"`. The `GET` method is used to transport the data from the HTML form to the Flask app.

The HTML code is saved as `index.html` file under the directory `Iris-API/templates/index.html`.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Flower Variety</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.9.0/css/bulma.min.css">
    <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
    <style>
        html{
            overflow: hidden;
        }

        body{
            position: absolute;
            width: 100%;
            height: 100%;
            margin: 0;
            padding: 0;
        }

        #login-form-container{
            position: absolute;
            width: 100%;
            height: 100%;
            display: flex;
            align-items: center;
            justify-content: center;
        }
    </style>
</head>
```

```
<body>
    <div id="login-form-container">
        <form action="predict" method="GET">
            <div class="card" style="width: 400px">
            <div class="card-content">
                <div class="media">
                <div class="is-size-4 has-text-centered">Flower Variety Classification</div>
                </div>
                <div class="content">

                <div class="field">
                    <p class="control">
                    Sepal Length: <input class="input" type="number" value='0.00' step='0.01' name="slen" id="slen">
                    </p>
                </div>

                <div class="field">
                    <p class="control">
                    Sepal Width: <input class="input" type="number" value='0.00' step='0.01' name="swid" id="swid">
                    </p>
                </div>

                <div class="field">
                    <p class="control">
                    Petal Length: <input class="input" type="number" value='0.00' step='0.01' name="plen" id="plen">
                    </p>
                </div>

                <div class="field">
                    <p class="control">
                    Petal Width: <input class="input" type="number" value='0.00' step='0.01' name="pwid" id="pwid">
                    </p>
                </div>

                <div class="field">
                    <button class="button is-fullwidth is-rounded is-success">Submit</button>
                </div>
                </div>
            </div>
        </form>
    </div>
</body>
```

Figure: Head and Body of the home HTML page.

25.2.3 Creating a Flask App

Next, we will use the Flask framework to deploy the Machine Learning model locally. Note that Flask is not typically run from Jupyter Notebooks, and therefore we will save the code from this section as a Python script `app.py` and we will run the script from the command line in a terminal.

Let's first import the `model` module that we created in the previous section. Next, we import the `flask` library, as well as the related libraries `request` (for accessing data sent with HTTP requests), and `render_template` (for generating HTML pages by combining templates with data).

Building the Flask server begins with creating an instance of the Flask application, which we assigned to the name `app`. The argument `__name__` associates the instance with the name of the current module, and Flask uses it to locate the root path of the application. The `template_folder` argument specifies the directory where Flask will look for HTML templates for rendering the webpages for input parameters and model output.

[ ]:
```
import model # Import the python file containing the ML model
from flask import Flask, request, render_template # Import flask libraries
```

(continues on next page)

(continued from previous page)

```
# Initialize the flask class and specify the templates directory
app = Flask(__name__, template_folder="templates")
```

In Flask, **routes** define the actions that the web API should take when a specific URL is accessed by the end-user in a web browser. Routes in Flask are defined using the `@app.route` decorator. This decorator is associated with a function that specifies the actions to a particular URL. In this example, we will define two routes: route `'/'` that corresponds to the home page where the end-user enters the input values for iris flowers, and route `'/predict'` that sends the input values to the model for inference and displays the model's prediction.

Therefore, in the next cell we set the default route of the Flask application using `@app.route('/')`. The route `'/'` in Flask is the root or home route. When we access the root URL of the application (e.g., `http://127.0.01:5000/`), the function `index_view` will be called, and it will render the HTML page `'index.html'` in the web browser. The `render_template()` is a Flask function that takes an HTML file as an argument and processes it for sending as the response to the end-user's request. This is a common way of creating a simple home page for a web API.

[ ]:
```
# Default route set as 'index'
@app.route('/')
def index_view():
    return render_template('index.html') # Render index.html
```

Similarly, in the next cell we create a separate Flask route for the Machine Learning model, that accepts inputs for the model, makes a prediction, and renders the results in a webpage.

Specifically, we use the `/predict` route with the method `'GET'`. This route will retrieve the data from the fields on the `index.html` webpage through a GET request. For this purpose, we use `request.args.get()` to extract the data from each of the input fields in the form, using the name attributes for the petal and sepal dimensions.

The retrieved inputs `sepal_len`, `sepal_wid`, `petal_len`, `petal_wid` are then passed to the `classify()` method of the Machine Learning model to make a prediction about the variety of the iris flower. The result from the model's prediction is assigned to the variable `variety`.

The variable `variety` is next passed to the HTML file `output.html` to render the predicted variety of flowers and display it in a new webpage. For this, we used the function `render_template(filename, arguments)` where we specified the HTML file along with the predicted variable `variety`.

Finally, if an exception occurs during the execution of the `try` block (e.g., an error in model prediction or input parameters), the message `'Error'` will be returned.

[ ]:
```
# Route 'predict' accepts GET request
@app.route('/predict', methods=['GET'])
def predict_class():
    try:
        sepal_len = request.args.get('slen') # Get parameters for sepal length
        sepal_wid = request.args.get('swid') # Get parameters for sepal width
        petal_len = request.args.get('plen') # Get parameters for petal length
        petal_wid = request.args.get('pwid') # Get parameters for petal width

        # Get the output from the classification model
        variety = model.classify(sepal_len, sepal_wid, petal_len, petal_wid)

        # Render the output in new HTML page
        return render_template('output.html', variety=variety)
    except:
        return 'Error'
```

To run the Flask webserver, we use the `app.run()` method as shown below. The `debug=True` argument enables the debug mode, which provides additional information in case of errors.

```
[ ]: # Run the Flask server
if(__name__=='__main__'):
    app.run(debug=True)
```

The code for Flask is saved as `app.py` under the directory `IRIS-API/app.py`. We will use it later to run the web application.

25.2.4 Output Webpage

We need to also create another even simpler HTML page to display the output of the classification.

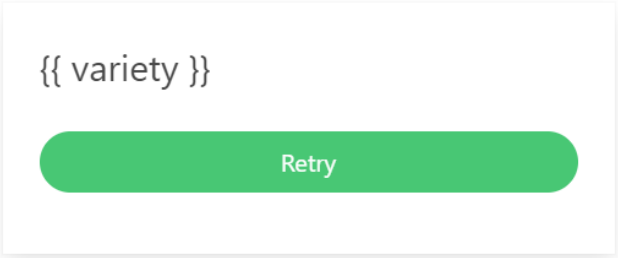


Figure: Output webpage to present the model's prediction to end-users.

The body of the code is shown below, and it is saved as `output.html` in the directory `./templates/output.html`. In the code `{{ variety }}` specifies the argument that was passed to get rendered in the new HTML file.

```
<body>
    <div id="login-form-container">
        <div class="card" style="width: 400px">
            <div class="card-content">
                <div class="media">
                    <div class="is-size-4 has-text-centered">{{ variety }}</div>
                </div>
                <form action="home">
                    <div class="field">
                        <button class="button is-fullwidth is-rounded is-success">Retry</button>
                    </div>
                </form>
            </div>
        </div>
    </div>
</body>
```

Figure: Body of HTML file for the output webpage.

25.2.5 Run the Web Application

To deploy the application, we will execute the command `python app.py` from the terminal. Don't forget to first change the directory to `Iris-API` where the file `app.py` is saved.

The output is shown in the figure below. We can see that there are warning messages with recommendations for changes in the code, but these warnings do not affect the functionality of the app. There is also a warning that this is a development server, which is primarily intended fro testing and debugging purposes. If we would like to deploy the web application in production, we will need to use WSGI (Web Server Gateway Interface) server that is designed to handle a large number of requests and is optimized fro performance and scalability.

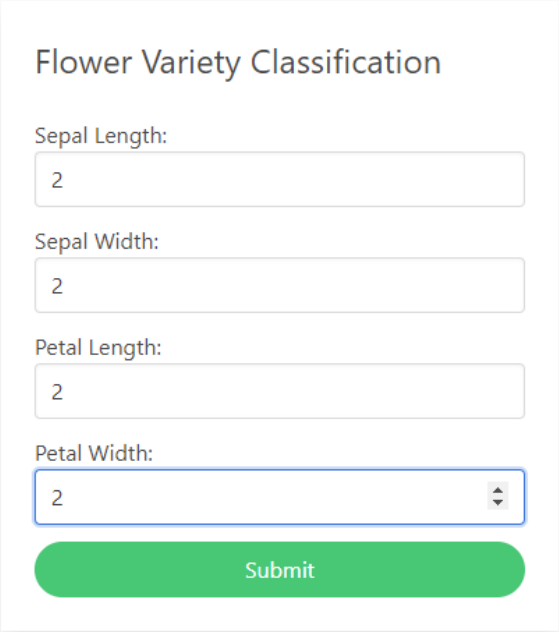
We can also see that the server is running in debug mode on `http://127.0.01:5000/` (given with the IP Address:Port).

To start the application on the local machine, we can either click on the URL `http://127.0.01:5000/`, or we can copy and paste the URL into a web browser.

```
PS C:\Users\vakanski\Documents\Codes\2023 Codes\Python for Data Science Course\Lectures_2023\Theme_4-Model_Deployment\Lecture_25-Deploying_to_Web\Iris-API> python app.py
C:\Users\vakanski\Anaconda3\Lib\site-packages\sklearn\preprocessing\_label.py:114: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please chan
ge the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
C:\Users\vakanski\Anaconda3\Lib\site-packages\sklearn\linear_model\_logistic.py:460: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Figure: Run the application.

The following are screenshots of the application for one example and the predicted iris class is Setosa.



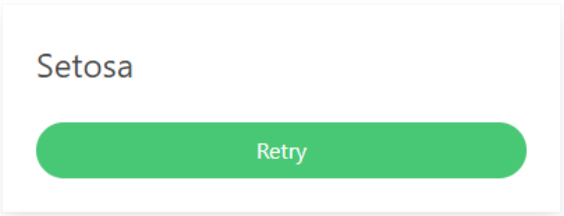


Figure: Entered inputs and predicted class.

The folder structure is as follows:

```
Iris-API
    ├── iris.csv
    ├── model.py
    ├── app.py
    ├── templates
    │   ├── index.html
    │   ├── output.html
```

7.25.3 25.3 Deploying a Model for MNIST Digits Classification

In this section we will create another web API for classification of MNIST digits. Differently from the previous section, we will use a Convolutional Neural Network model, as well as we will save the trained model and use it for prediction. The code in mostly based on this section from the following blog: Deploying Deep Learning Models Part 1: Preparing the Model.

25.3.1 Model Training and Saving

We will first load the MNIST dataset, and we will create a simple Convolutional Neural Network for classification of the digit images in the dataset.

[1]:
```
import tensorflow
from tensorflow import keras
from keras.datasets import mnist
from keras.models import Model
from keras.layers import Input, Dense, Dropout, Flatten, Conv2D, MaxPooling2D
```

[2]:
```
# Load the data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

[3]:
```
# Scale the images to the range [0,1]
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# Print the shape and number of samples
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

```
x_train shape: (60000, 28, 28)
60000 train samples
10000 test samples
```

[5]:
```
# Define the layers in the model
inputs = Input(shape=(28, 28, 1))
conv1a = Conv2D(filters=32, kernel_size=3, padding='same')(inputs)
conv1b = Conv2D(filters=64, kernel_size=3, padding='same')(conv1a)
pool1 = MaxPooling2D()(conv1b)
flat = Flatten()(pool1)
dense1 = Dense(128, activation='relu')(flat)
dropout1 = Dropout(0.5)(dense1)
outputs = Dense(10, activation='softmax')(dropout1)

# Define the model with inputs and outputs
model = Model(inputs, outputs)
```

[6]:
```
# Compile and train the model
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

model.fit(x_train, y_train, batch_size=128, epochs=5, verbose=0)
```

[6]:
```
<keras.src.callbacks.History at 0x1bcccb76090>
```

[7]:
```
# Evaluate on test dataset
score = model.evaluate(x_test, y_test, verbose=0)
print('Test accuracy:', score[1])
```

```
Test accuracy: 0.9840999841690063
```

Next, we will save the model using the Keras-TensorFlow SaveModel format. We will use the saved model to make predictions in the web application without retraining the model, since training the model takes time and there is no need to train a new model every time the end-users need to make predictions.

[8]:
```
# save the model to a file
model.save("mnist_model.keras")
```

To make predictions, we can load the model and compile it, as in the following cell.

[9]:
```
from tensorflow.keras.models import load_model

# load the weights into new model
loaded_model = load_model("mnist_model.keras")

# Compile and evaluate loaded model
loaded_model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=[
↪'accuracy'])

# Evaluate on test dataset
score = loaded_model.evaluate(x_test, y_test, verbose=0)
print('Test accuracy:', score[1])
```

```
Test accuracy: 0.9840999841690063
```

25.3.2 Creating a Flask App

To deploy the model we will create the `app.py` file, similarly to the Flask file in the previous section.

We will define again two routes in the Flask app:

- An index (home) page route `'/'`, for the users to draw a digit image.
- A predict route `'/predict'`, to predict the class of the digit with the saved model.

In the next cell, we first import the required libraries, and an `app` object is created that is an instance of the Flask class.

[ ]:
```
from flask import Flask, render_template, request
from tensorflow.keras.models import load_model
from PIL import Image
import numpy as np
import re
import base64

# Initialize the flask class and specify the templates directory
app = Flask(__name__, template_folder="templates")
```

Similar to the previous example, we define the home page route using the `@app.route` decorator, and pass the URL of the main page of the web application `index.html`.

[ ]:
```
# Default route set as 'index'
@app.route('/')
def index_view():
    return render_template('index.html')
```

The `'/predict'` route is shown next, which gets the image of a digit drawn by the user, and it first applies image resizing and reshaping, and afterward the saved model is used to predict the digit class. Output is the `response` variable that returns a string of the predicted class.

[ ]:
```
# Route 'predict'
@app.route('/predict', methods=['GET', 'POST'])
def predict_class():

    # Get the drawn image
    imgData = request.get_data()

    # Convert the image into a png file
    convertImage(imgData)

    # Open the png file
    x = Image.open('output.png')

    # Resize the image to 28x28 pixels
    x = x.resize((28,28))

    # Invert the image into black background and white foreground
    x = np.invert(x)
```

(continues on next page)

(continued from previous page)

```
    # Reshape into a batch with one image
    x = x[:,:,0].reshape(1, 28, 28, 1)

    # Scale to the range [0,1]
    x = x/255

    # Make a prediction
    out = model.predict(x)

    # Get the predicted digit
    response = np.array_str(np.argmax(out, axis=1))

    return response
```

And finally, we run the app in debug mode.

```
[ ]: # Run the Flask server
if __name__ == '__main__':
    app.run(debug=True)
```

The organization of the directory of the web application is shown below. Beside the `app.py`, the main directory also contains the saved model `mnist_model.keras` and the image drawn by the users saved as `output.png`. The `templates` sub-directory contains the `index.html` and `draw.html` pages. In the `static` folder, a JavaScript file `index.js` is used to render a canvas for drawing digits by the users, and `style.css` is a stylesheet for the web application.

```
MNIST-API
    ├── app.py
    ├── mnist_model.keras
    ├── output.png
    ├── templates
    │   ├── index.html
    │   ├── draw.html
    ├── static
    │   ├── index.js
    │   ├── style.css
```

25.3.3 Run the Web Application

We can run the application using `python app.py`, and we will click on the URL `http://127.0.0.1:5000/` to access the web API shown below. We can use the mouse pointer to draw digits, and the `Predict` button will display the Predicted Output by the model. The `Clear` button allows to draw again and make a new prediction.

```
PS C:\Users\vakanski\Documents\Codes\2023 Codes\Python for Data Science Course\Lectures_2023\Theme_4-Model_Deployment\Lecture_25-Deploying_to_Web\MNIST-API> python app.py
2023-11-29 11:38:01.835956: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in performance-critical o
perations.
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
Loaded Model from disk
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with watchdog (windowsapi)
```

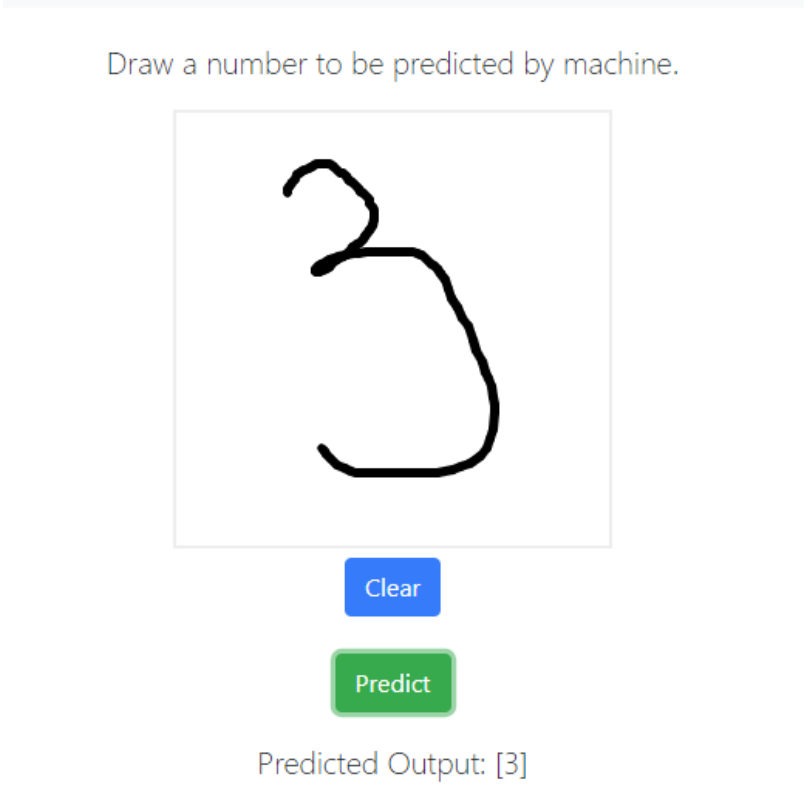


Figure: Run the web application.

7.25.4 Appendix

The material in the Appendix is not required for quizzes and assignments.

Deploying a Model with Fast API

This section explains how to develop a RESTful API for classification of Iris flowers with the Fast API library.

Before we try to develop the API, we need to install the required libraries with `pip install, fastapi, python-multipart, uvicorn`.

The code for the `app.py` server in Fast API is shown below.

First, we import modules from Fast API for handling HTTP requests and exceptions, and for template rendering.

Next, we instantiate the `app` web application from the `FastAPI()` class. And we define the directory with the HTML templates. Fast API uses Jinja2 for template rendering, allowing to create dynamic HTML pages.

Similar to the Flask app, in Fast API we use the `@app.get('/')` decorator to associate the HTTP GET method with the default home page of the application. In Fast API, the `Request` class is used to handle the incoming requests, where `request` is an instance that contains the information about the HTTP request. The `index_view` function will render a template of the `index.html` file and will pass the `request` object to it.

The `'/predict'` route will handle the GET request. The `predict_class` function retrieves the input values `slen`, `swid`, `plen`, and `pwid` and calls `model.classify()` to predict the variety of the Iris flower. If there are no errors, the function will render a template of the `output.html` file and will pass the `variety` to be displayed to the user.

Finally, the web application is run using the `uvicorn` server. UVicorn is the default server for Fast API in Python.

The code for the Fast API app is similar to the Flask app, but there are several differences in the syntax and the implementation.

[ ]:
```
# Import libraries
from fastapi import FastAPI, Request, Depends, Form, HTTPException
from fastapi.templating import Jinja2Templates
from typing import Optional
import model

# Initialize the FastAPI class and specify the templates directory
app = FastAPI()
templates = Jinja2Templates(directory="templates")

# Default route set as 'index'
@app.get('/')
def index_view(request: Request):
    return templates.TemplateResponse("index.html", {"request": request})

# Route 'predict' accepts GET request
@app.get('/predict')
def predict_class(request: Request,
                  slen: Optional[float] = None,
                  swid: Optional[float] = None,
                  plen: Optional[float] = None,
                  pwid: Optional[float] = None):
    try:
        # Get the output from the classification model
        variety = model.classify(slen, swid, plen, pwid)

        # Render the output in a new HTML page
        return templates.TemplateResponse("output.html", {"request": request, "variety":␣
↪variety})
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"An error occurred: {str(e)}")

# Run the FastAPI server
if __name__ == '__main__':
    import uvicorn
    uvicorn.run(app)
```

Deploying a Model with Django

The corresponding code for developing a RESTful API for classification of Iris flowers with the Django library is shown below.

The code is similar to the apps in Flask and Fast API. The functions `index_view` and `predict_class` define the rendering of the home page and the predicted outputs by the model.

The URL patterns define that the root path maps to the `index_view` function, and the `'predict'` path maps to the `predict_class` function.

The last part of the code defines the configuration settings and the execution of the server with `app.py runserver`.

[ ]:
```
from django.shortcuts import render
from django.http import HttpResponse
from django.urls import path
import model

# Default route set as 'index'
def index_view(request):
    return render(request, 'index.html')

# Route 'predict' accepts GET request
def predict_class(request):
    try:
        slen = float(request.GET.get('slen'))
        swid = float(request.GET.get('swid'))
        plen = float(request.GET.get('plen'))
        pwid = float(request.GET.get('pwid'))

        # Get the output from the classification model
        variety = model.classify(slen, swid, plen, pwid)

        # Render the output in a new HTML page
        return render(request, 'output.html', {'variety': variety})
    except Exception as e:
        return HttpResponse(f"An error occurred: {str(e)}", status=500)

# URL patterns define which function is called for specific URL paths
urlpatterns = [path('', index_view),
    path('predict/', predict_class)]

# Django settings and execution
if __name__ == '__main__':
    from django.conf import settings

    # Define the templates directory and debug mode
    settings.configure(
        DEBUG=True,
        ROOT_URLCONF=__name__,
        TEMPLATES=[
            {
                'BACKEND': 'django.template.backends.django.DjangoTemplates',
                'DIRS': ['templates'],
                'APP_DIRS': True,
            },
        ],
    )

    # Execute the server from the command line
    import django
    django.setup()

    from django.core.management import execute_from_command_line

    execute_from_command_line(['app.py', 'runserver'])
```

7.25.5 References

1. Deploying Machine Learning Models using Flask, by Srishilesh P S, available at https://www.section.io/engineering-education/deploying-machine-learning-models-using-flask/.
2. Deploying Deep Learning Models Part 1: Preparing the Model, by Vihar Kurama, available at: https://blog.paperspace.com/deploying-deep-learning-models-flask-web-python/.
3. Python and REST APIs: Interacting With Web Service, by Jason Van Schooneveld, available at https://realpython.com/api-integration-in-python/.
4. Getting Started with RESTful APIs and Fast API, by Sunil Kumar Dash, available at https://www.analyticsvidhya.com/blog/2022/08/getting-started-with-restful-apis-and-fast-api/.

BACK TO TOP

7.26 Lecture 26 - Deploying Projects to the Cloud

7.26.1 26.1 Data Science using Cloud Computing

Cloud Computing, also referred to as the Cloud, delivers services hosted over a network, which can include data analytics, storage, databases, networking, and other services. The service is often in the form of a *public cloud* offered to the public over the Internet by cloud service providers, or it can be a *private cloud* that is owned by an organization that maintains the services on a private network.

Public Cloud Computing services include Amazon Web Services, Google Cloud Platform, Microsoft Azure, IBM Cloud, and others.

In general, Cloud Computing services can be categorized as:

- *Infrastructure as a Service (IaaS)*: access to infrastructure, consisting of servers, virtual machines (VMs), storage, networks, or databases.
- *Platform as a Service (PaaS)*: access to a platform for developing, testing, delivering, and managing software applications, using infrastructure managed by the provider.
- *Software as a Service (SaaS)*: access to software applications, that are developed and managed by the provider using the provider's infrastructure.

The advantages of using Cloud Computing include convenient access to the latest computational resources (without the need to purchase hardware or software), access to structured environments (preinstalled libraries) for running tasks, pay for what you need only, ability to quickly scale projects, improved efficiency by relying on infrastructure hosted and managed by the cloud provider, etc.

Cloud Computing is especially important for managing Data Science projects, which often require access to GPUs and large compute resources, storing large amounts of data, access to databases, deploying solutions for access by end-users, and similar. In addition, most Cloud providers have developed some form of AutoML tools that enable organizations without Data Science expertise to implement data analytics workflows into their projects.

This lecture is primarily based on a course Data Science for Beginners by Microsoft. The course has several lectures on deploying Data Science projects to the Cloud, as well as it has other lectures on Data Science in general.

7.26.2 26.2 Introduction to Azure Machine Learning

Microsoft's Azure Machine Learning is a cloud platform that provides a large number of products and services designed for handling various phases of Data Science projects. This includes capabilities for preparing and preprocessing data, training models, deploying models, and monitoring models in production. These capabilities can help to increase the efficiency of data scientists by automating many tasks and project pipelines. Understandably, the availability of cloud computing resources allows to easily scale projects and handle efficiently challenges related to processing big data and serving large a number of customers.

Important tools and services provided by Azure ML include:

- *Azure Machine Learning Studio*: framework for data engineering, model training, and deployment.
- *Azure Machine Learning Designer*: low-code ML framework that allows to drag-and-drop modules for building data science pipelines.
- *Azure Machine Learning SDK*: code-based environment for data science projects.
- *Data Labelling*: tools for automatic data labeling.
- *Machine Learning CLI (Command-Line Interface)*: allows managing Azure ML resources from the command line.
- *Automated Machine Learning (AutoML) User Interface*: tools to automate tasks in data science projects.
- *MLflow*: framework for tracking the performance of deployed models, and logging metrics and relevant indicators.

Azure ML allows using Jupyter Notebooks and has built-in integration with popular ML libraries like Scikit-Learn, TensorFlow, PyTorch, and others.

In this lecture, we will explore the different levels of functionality of Azure ML, ranging from the no-code AutoML, to full-code SDK, and working with our own custom models.

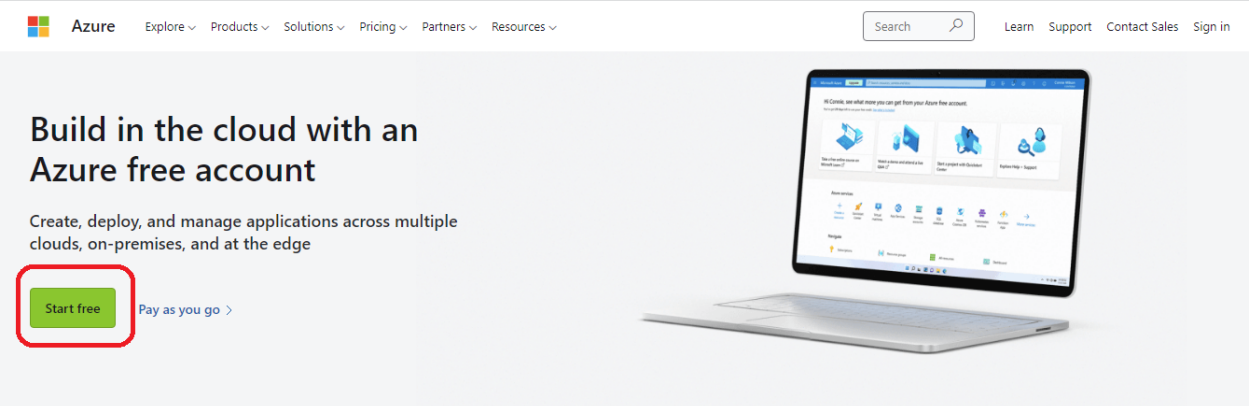
26.2.1 Azure Free Trial

Microsoft Azure has 30 days of free trial, which also can come with a $200 Azure credit that can be used within the 30 days.

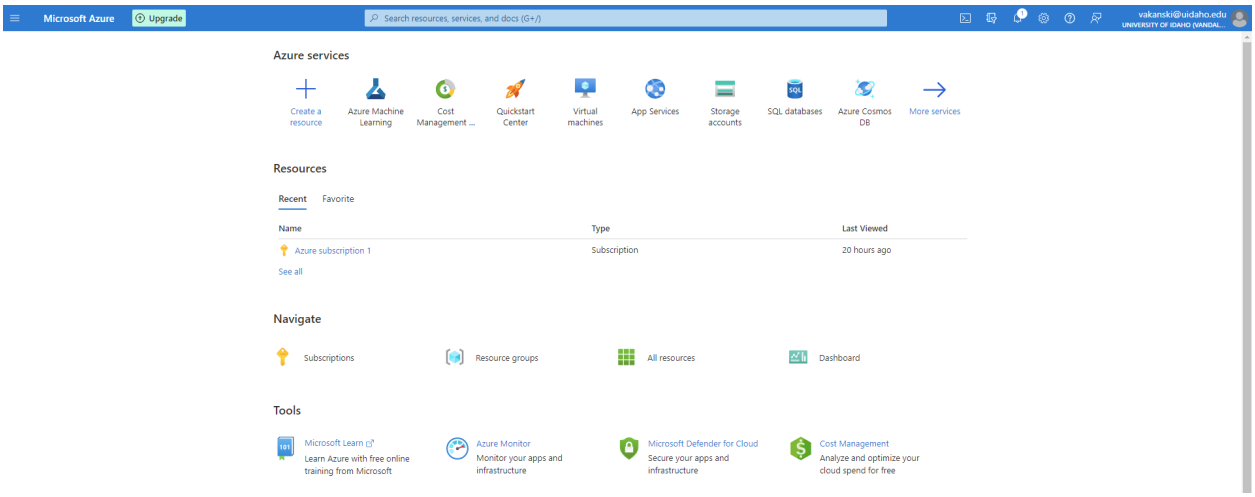
Also, Azure offers $100 yearly Azure credit to students.

In addition, the other Cloud providers typically offer some amount of credit to new users and students.

Follow the link to the Microsoft Azure webpage and select the `Start free` button. This will prompt you to create an Azure account, and if you wish you can use your University of Idaho account to get access to Azure.



Once you create an account and get the subscription with $200 Azure credits, the home page should look similar to the following.

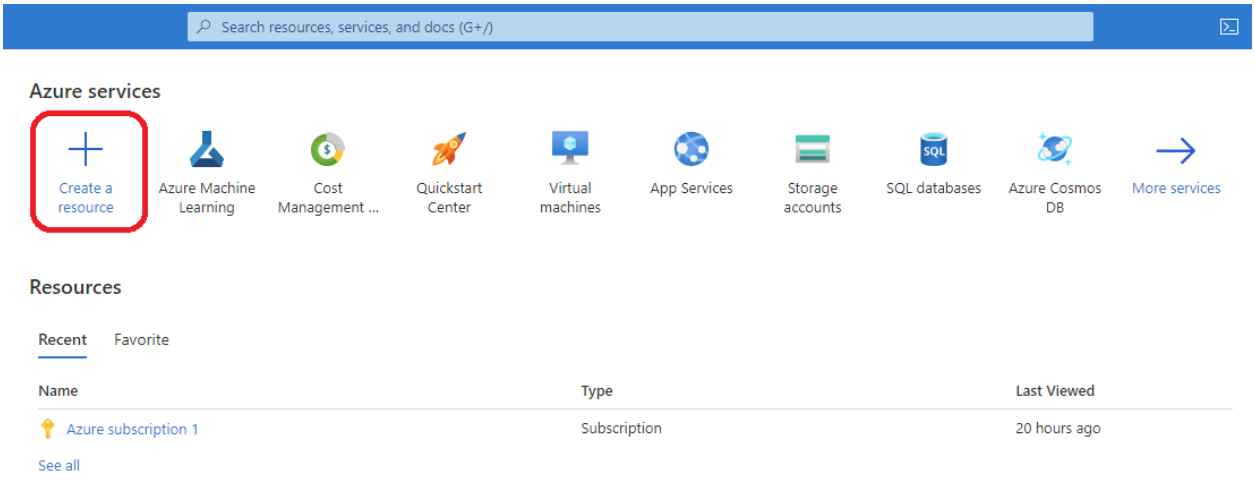


7.26.3 26.3 No-Code Azure ML

26.3.1 Creating a Workspace Resource

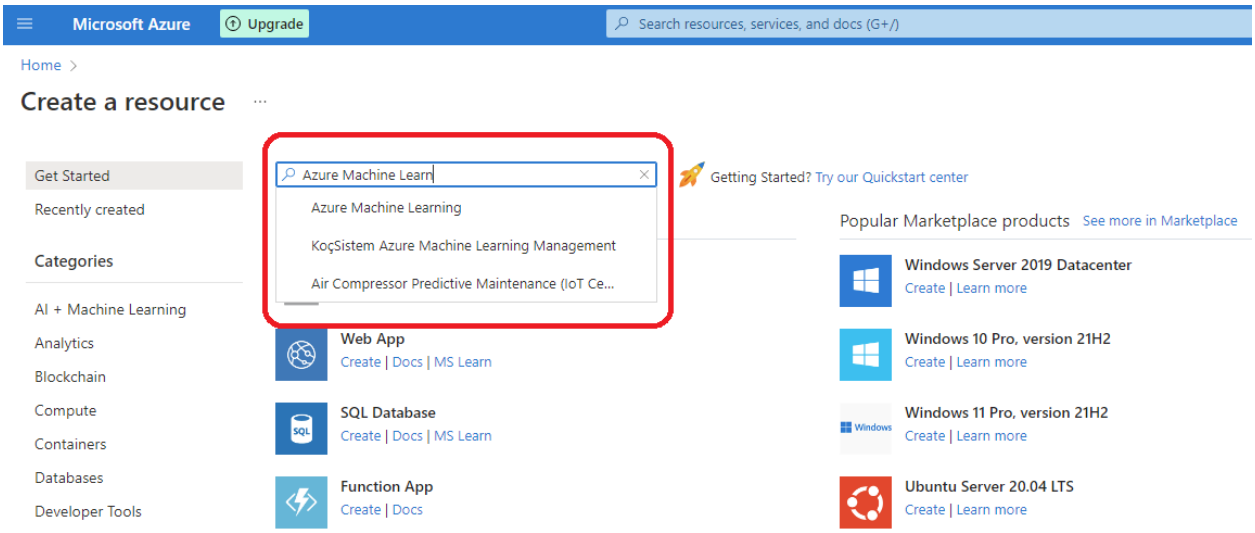
From the home page, we will need to first create a new **Resource** that will indicate what type of tools and services we will be using.

- Select `+ Create a resource`.



Azure will next display many popular services and resources.

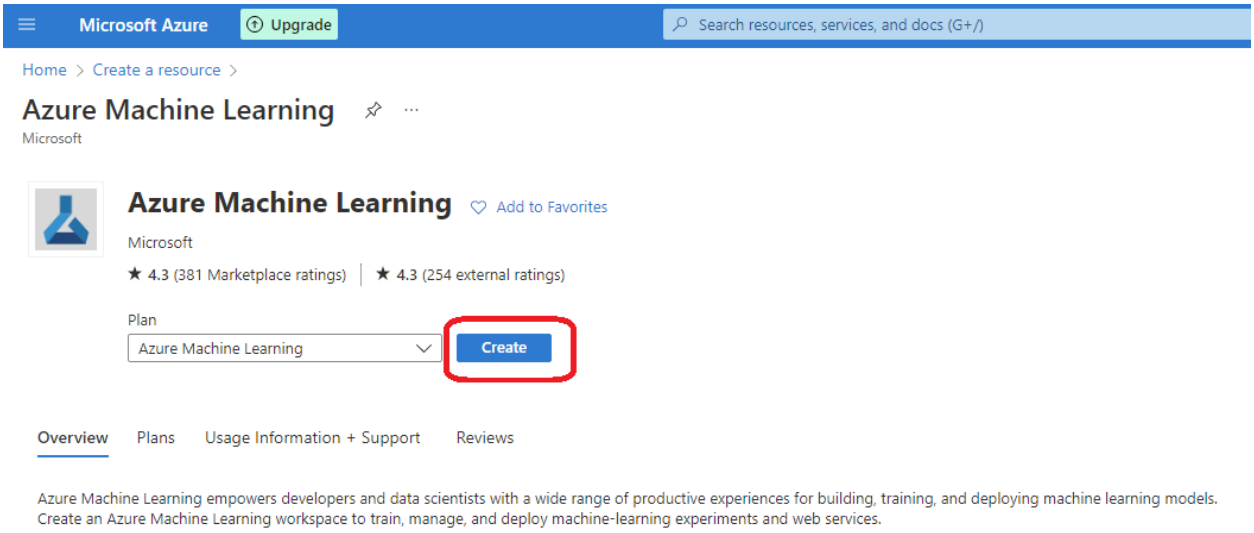
- In the search box write `Azure Machine Learning` and select it.



This will load the web page of Azure Machine Learning.

- Select `Create`.

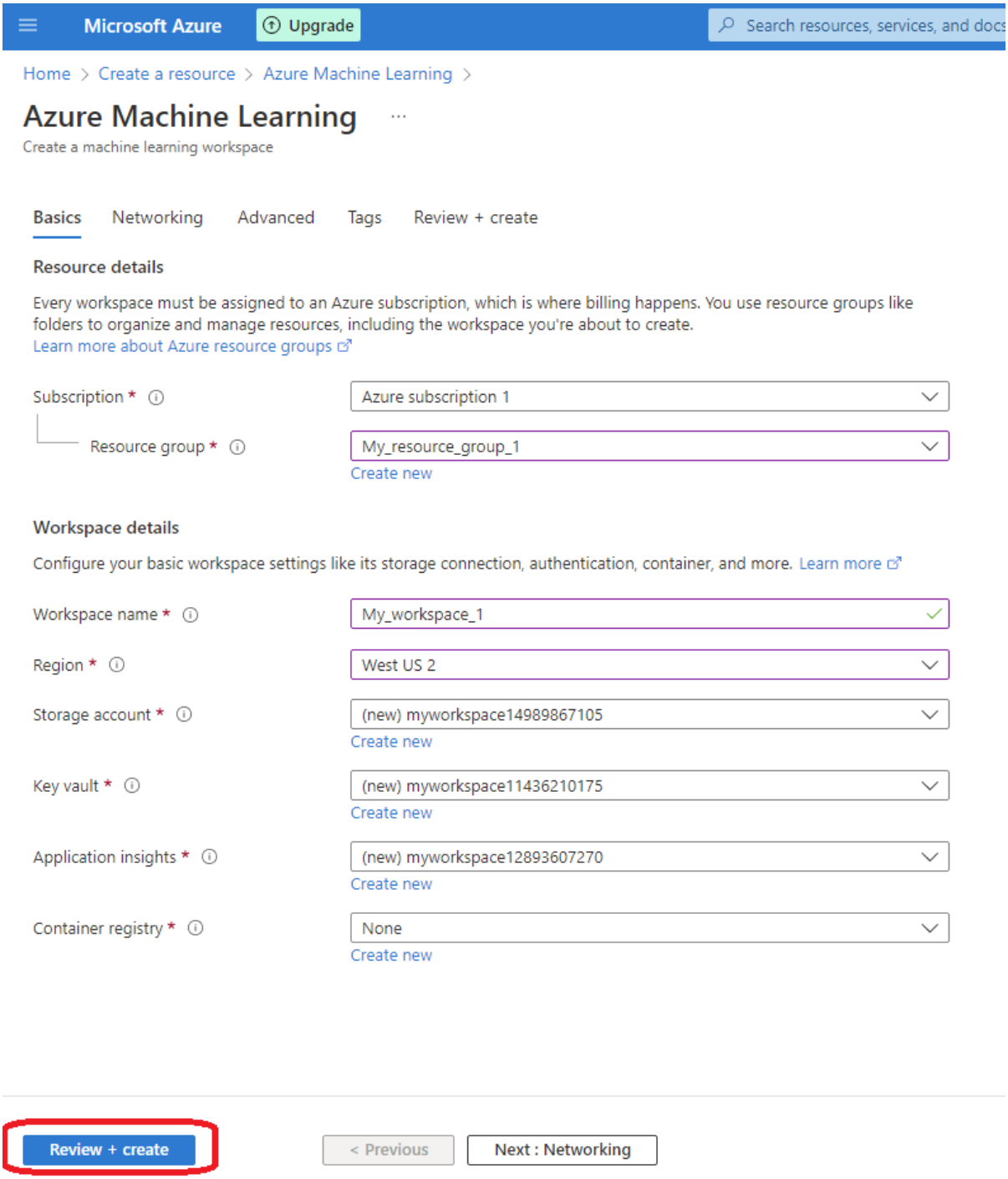
It will open a new page for `Azure ML Workspace resource`. The **Workspace** provides a place to work with machine learning models, and allows access to tools for training and deploying models. For instance, the Workspace will store information about training runs, such as logs of various metrics, it will provide access to the data and scripts, etc. And note also that when we are done with using Azure resources such as workspaces, we need to delete the resources, otherwise some costs can be incurred (e.g., even if we don't use the workspace to run a model, Azure may charge a fee for storing the data).



When we create a new Workspace, we need to fill in the information shown in the screenshot below.

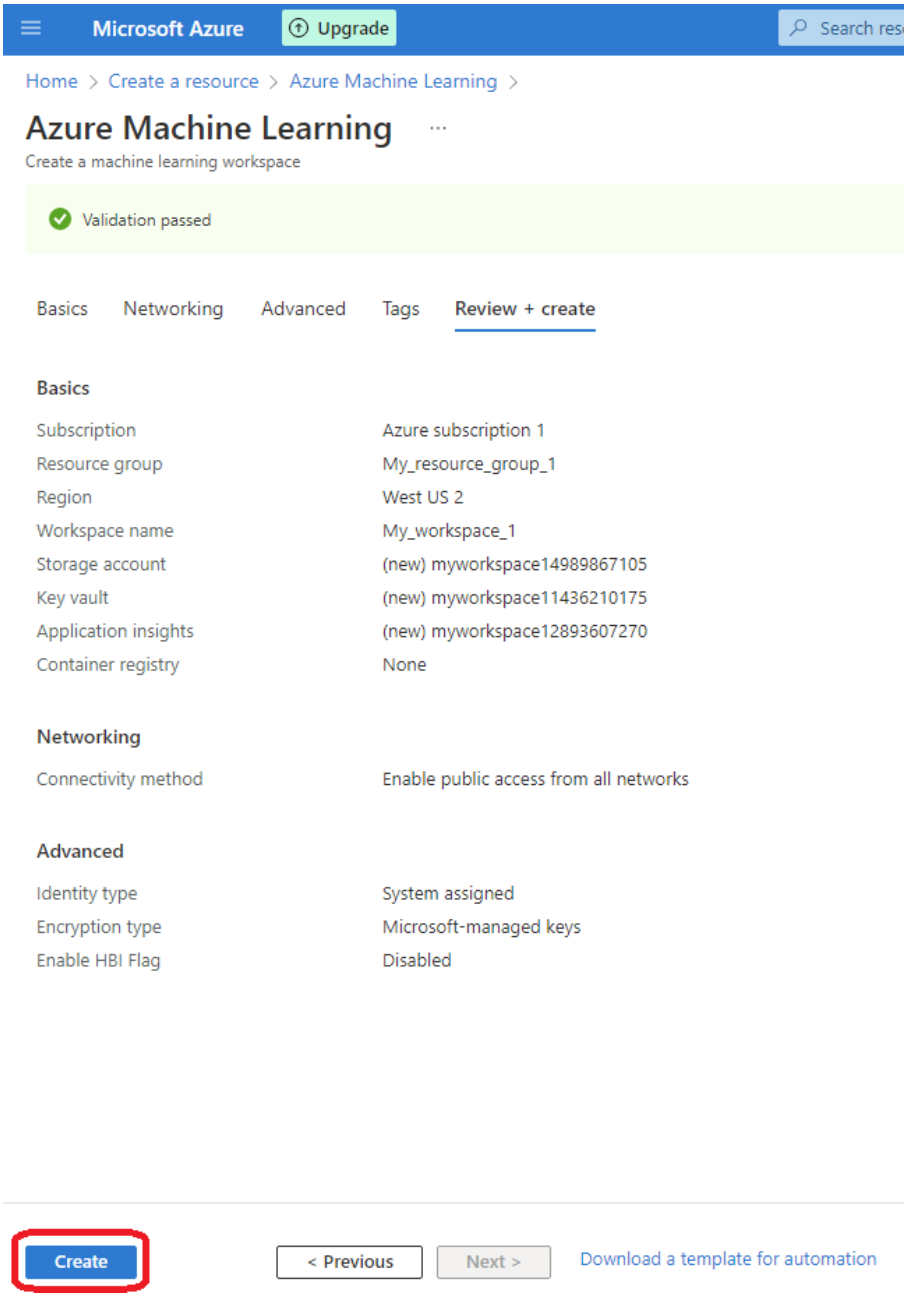
- Subscription: Azure subscription, e.g., the $200 Azure credits obtained with the free trial.
- Resource group: Assign a name for the resource group, or click on `Create new` to create a resource group.
- Workspace name: Assign a name for the workspace (e.g., perhaps a name that is related to the project).
- Region: Select the geographical region.
- Storage account: A new storage account will be created for the workspace for storing the data.
- Key vault: A new key vault will be created for the workspace for storing sensitive information.
- Application insights: A new application insights resource will be created for the workspace to store information about deployed models.
- Container registry: Leave it as None (it will be created automatically the first time the model is deployed).

After the information in all fields is entered, select `Review & create`.

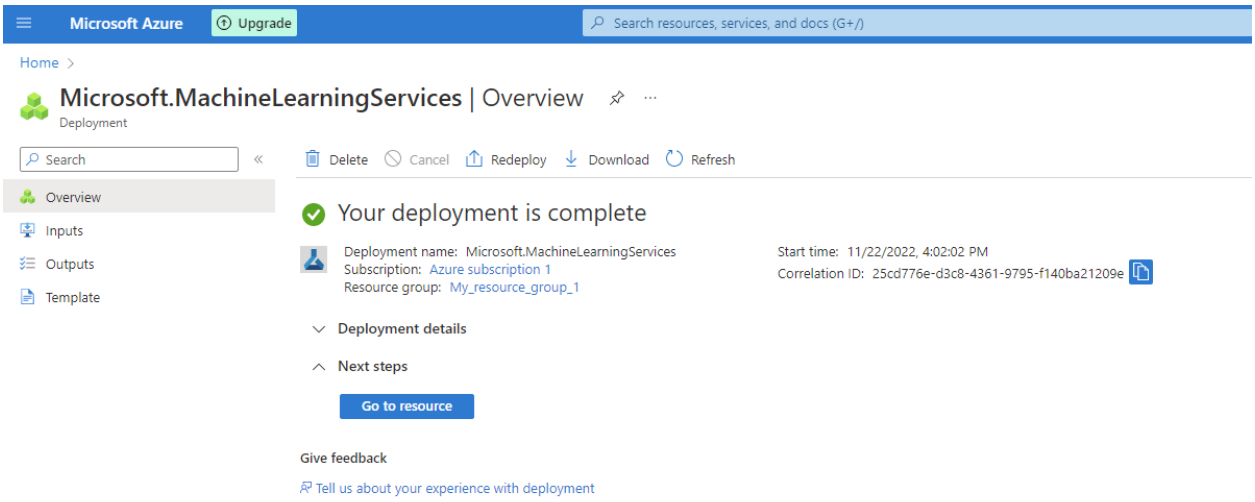


The next page will show the information that we entered and we will need to confirm that everything is correct.

- Select `Create`.



It may take a few minutes for the Workspace to be created. Once it is ready, the page will show that it is completed.

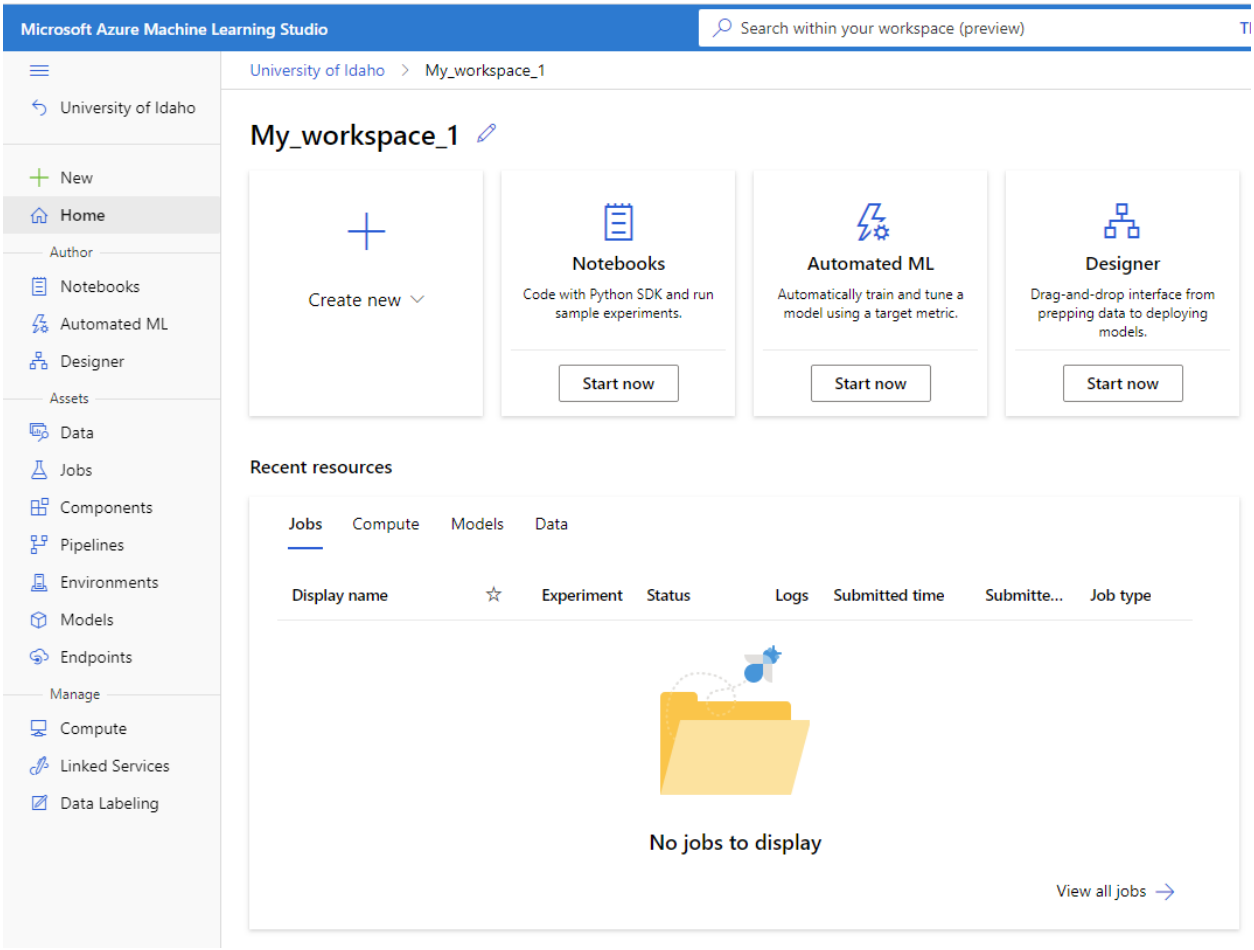


26.3.2 Using Azure ML Studio

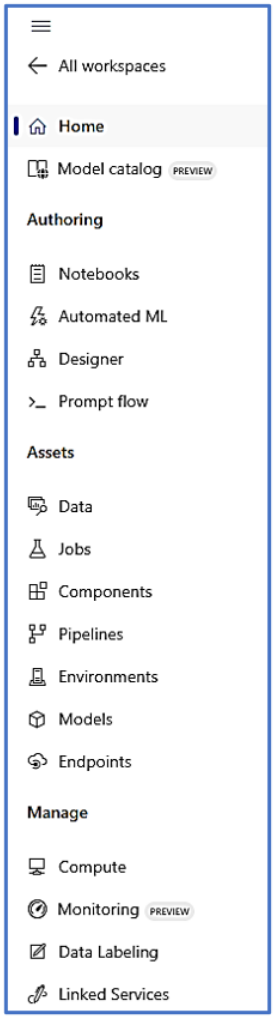
As we mentioned in the introductory section, **Azure Machine Learning Studio** is a framework for data engineering, model training, and deployment.

- Click on the following link to navigate to Azure ML Studio.

The interface of Azure ML Studio is shown below. On the top of the page, our workspace should be listed. In this case, the workspace that we just created and named `My_workspace_1` is shown.



The various modules that are available in Azure ML Studio are listed in the left-side menu. To see the names of the modules, click on the three horizontal lines in the upper left corner. A brief description of the modules is shown in the next figure. The modules allow to conveniently apply tools for managing different phases of data science projects from a single place.



26.3.3 Loading the Dataset

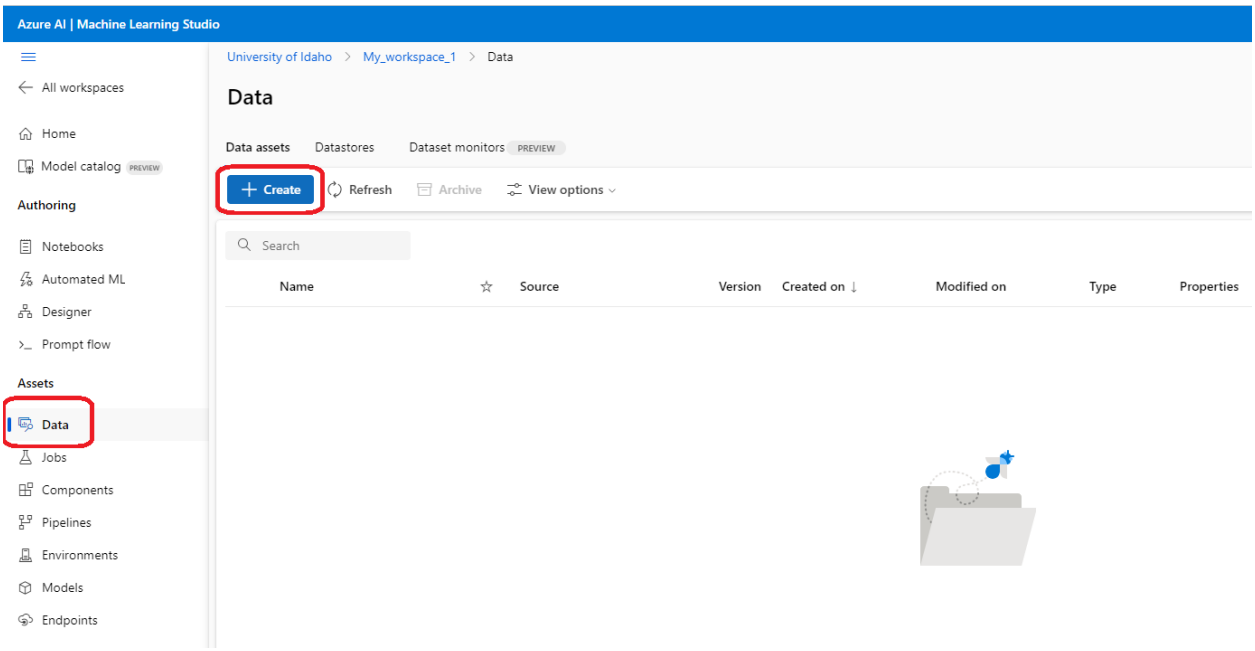
For demonstration purposes, we will use the Heart Failure Dataset, which we used before in this course, and contains 13 columns with information about 300 patients who may or may not have risk of heart failure. The `.csv` file with the records is available in the `data` folder with the other files for this lecture.

- Click on the `Data` module in the left-side menu in Azure ML Studio (see figure below).

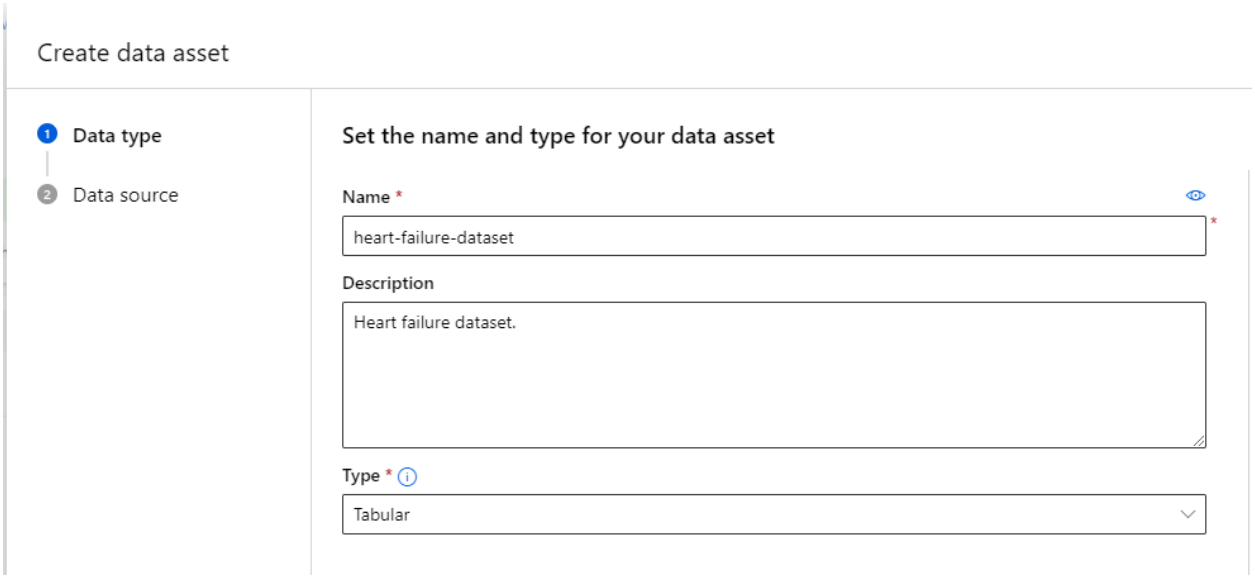
The Data section provides various tools for data management, and it allows to upload files or folders with data from a local machine, or provide links to web files (e.g., data from GitHub or Google Drive), load data from a list of open datasets collected by Microsoft (check Azure Open Datasets), or use files from a datastore. Datastores allow organizations that have many data files in different locations in Azure to link them together and organize them in a single view.

- Select `+ Create` to load the dataset.

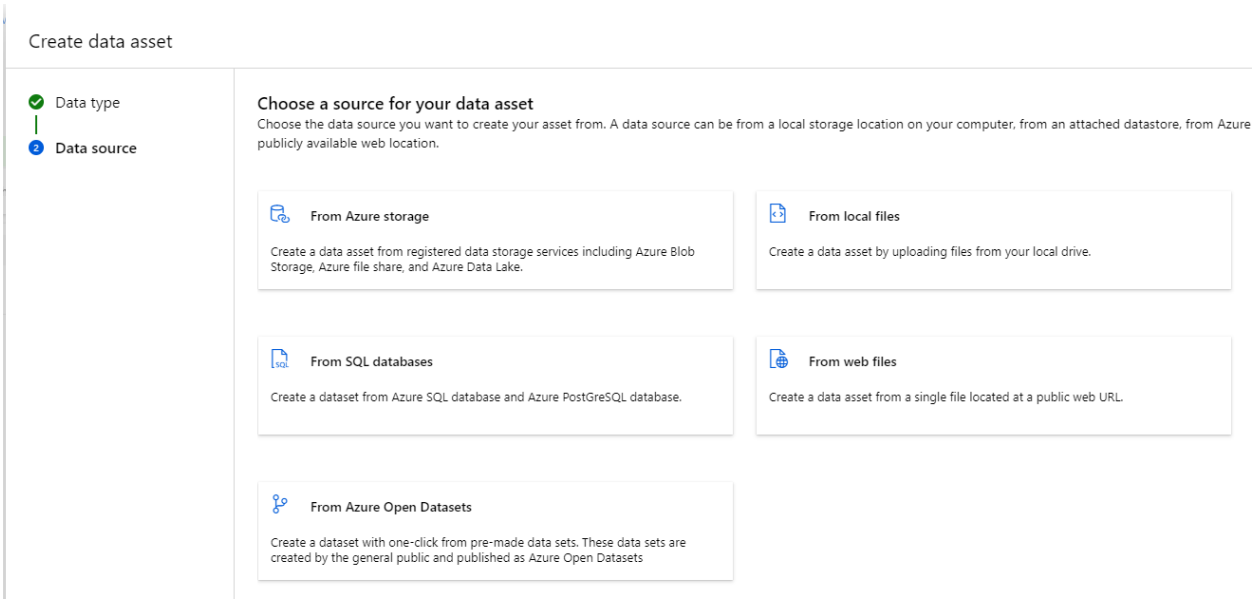
Azure ML Studio will guide us through several steps for creating the dataset.



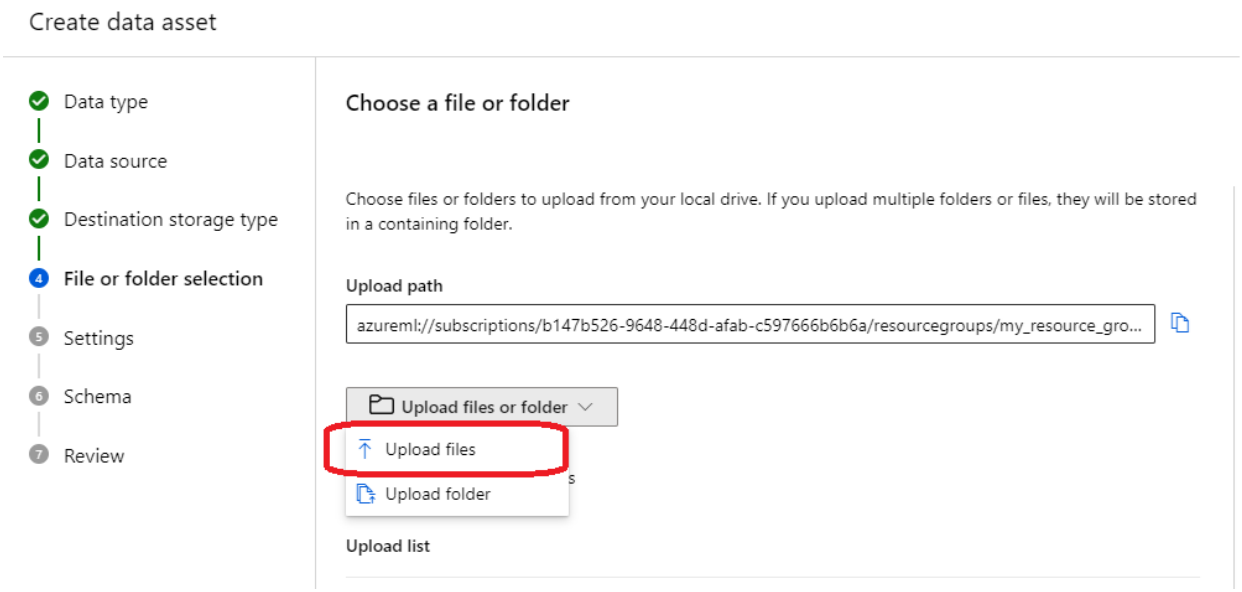
On the first page, we will need to enter a name for the dataset, a brief description, and also indicate whether the data is in tabular or other formats. After the information is entered, select `Next`.



Afterward, we will indicate that the dataset is saved on our computer and we will upload the dataset from local files.



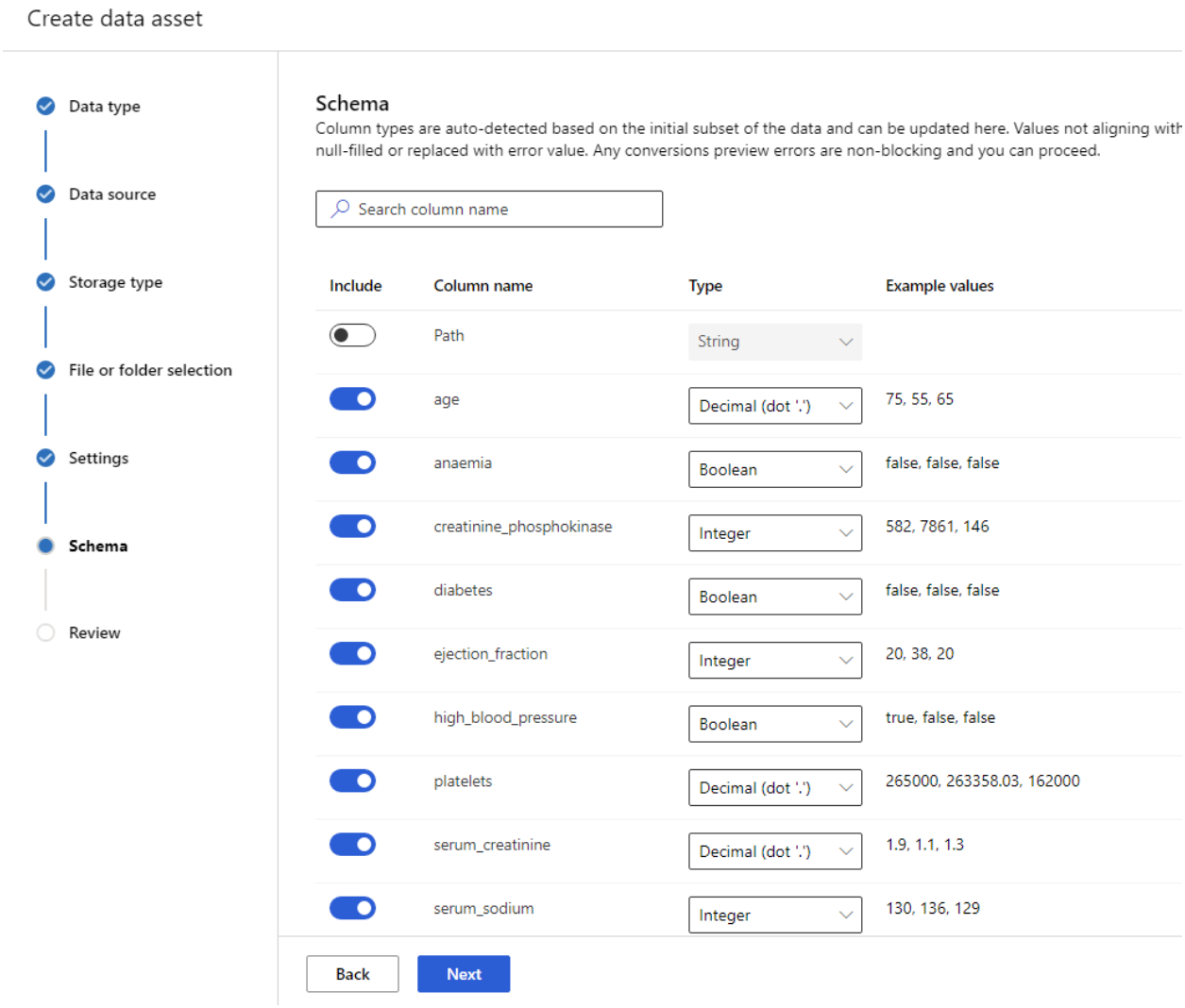
Select Next to skip Step 3 and get to Step 4 where from the dropdown menu we select `Upload` - `Upload files` and navigate to the directory where you have saved the csv file containing the heart failure records.



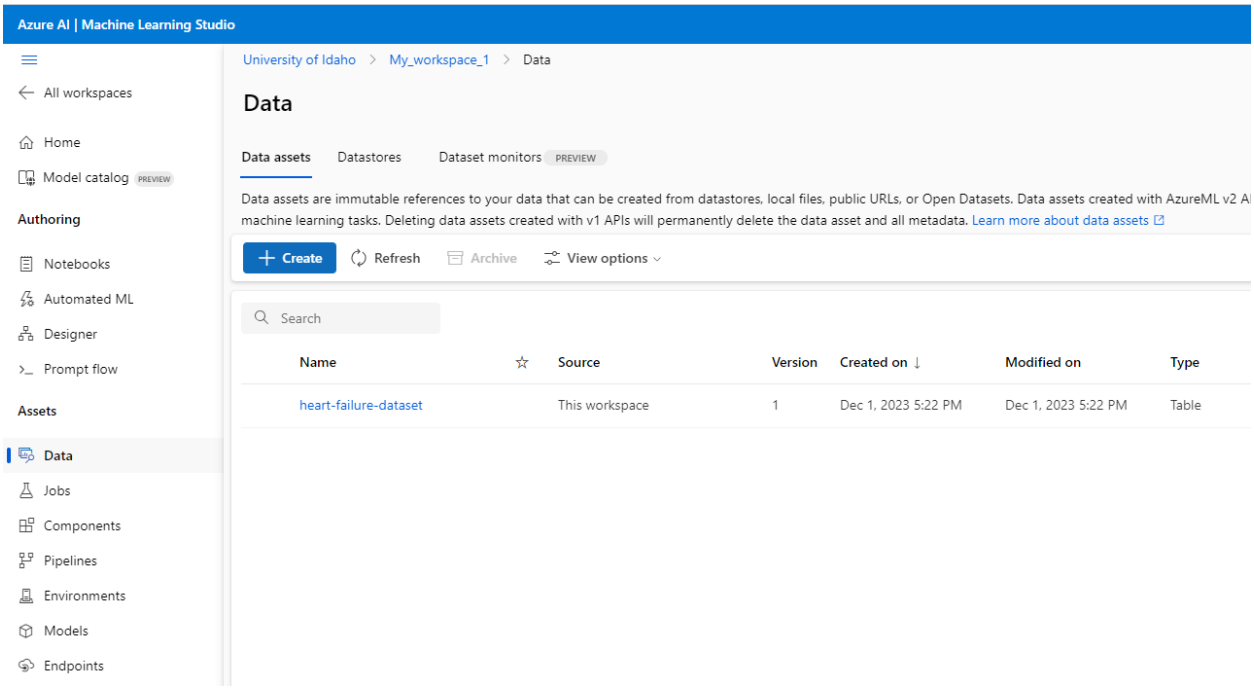
The next page will show the columns in the dataset. Select `Next` to go to the next page.

In the Schema page, we will change the data type to Boolean for the columns `anemia`, `diabetes`, `high blood pressure`, `sex`, `smoking`, and `DEATH_EVENT`.

Afterward, click `Next` and select `Create` to complete the creation of the dataset.



Now we can see that the dataset `heart-failure-dataset` is listed under the `Data assets` in our workspace.

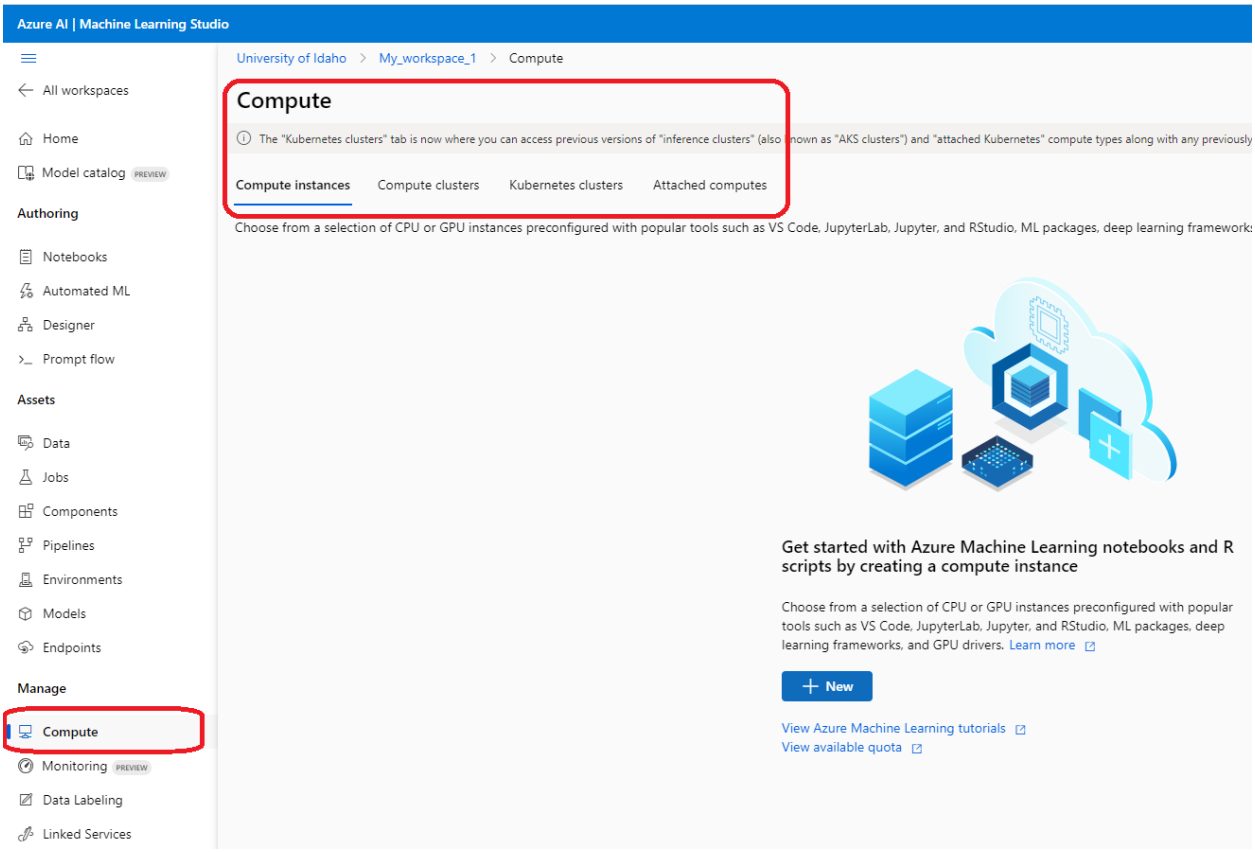


26.3.4 Creating a Compute Resource

We also need to use Compute Resources for our project to perform data preparation and processing, and to run the models. To create a compute resource, we will select `Compute` from the left-side menu.

We can see that the compute resources are categorized into four tabs:

- Compute Instances, are workstations for data and models; they involve creating a Virtual Machine (VM) and launching a notebook instance (e.g., compute resources to train a model are requested from the notebook).
- Compute Clusters, VMs for on-demand code processing (e.g., training a model using AutoML).
- Kubernetes Clusters, VMs that are orchestrated by Kubernetes.
- Attached Compute, links to existing Azure compute resources, such as Virtual Machines or Azure Databricks clusters.



For this task, we can use either a compute instance or compute cluster, so let's select the `Compute instances` tab.

- Click on the `+ New` button to create a new compute resource.
- Let's assign the name `heart-failure-compute` for the resource (see the figure below).

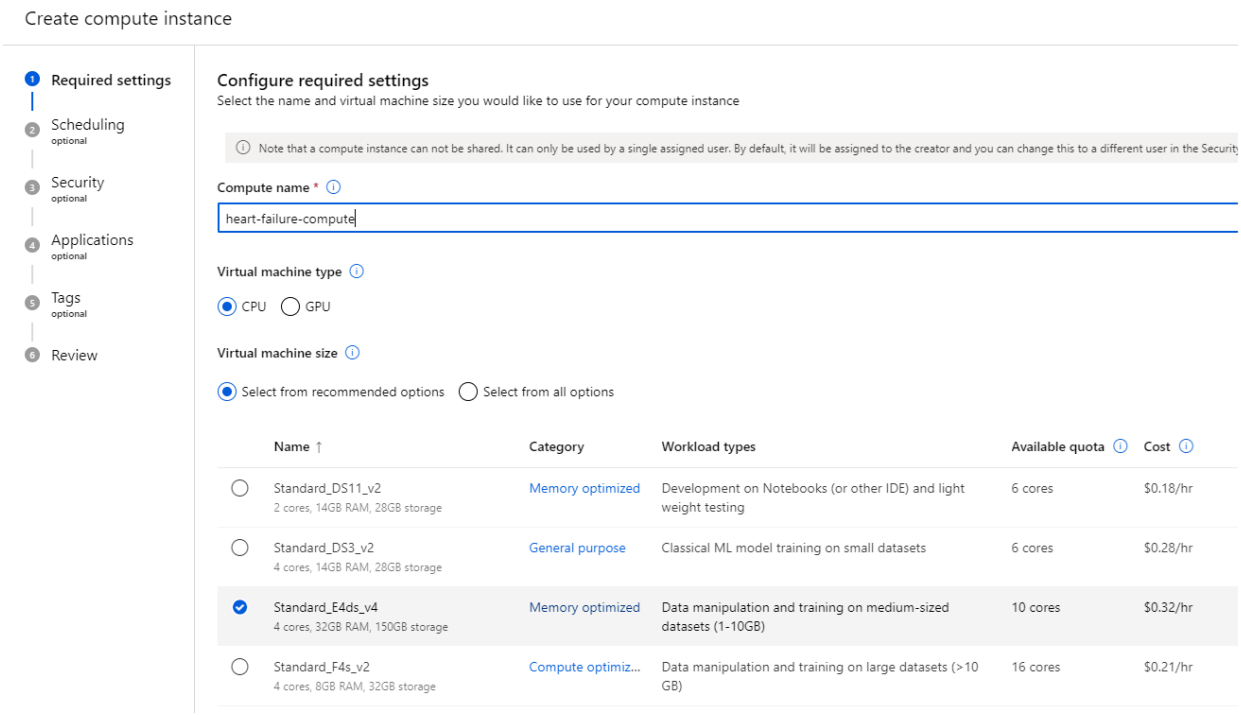
Selecting adequate compute resources for a project depends on several factors, which impose trade-offs between speed and cost.

1. *CPU versus GPU*: CPUs are less expensive, but also less powerful especially for training deep learning models. GPUs are more expensive, but they provide efficient parallel computing, and are often necessary for training deep learning models.
2. *Cluster size*: larger clusters are more expensive, but faster in completing tasks. For smaller tasks that don't take too long, it may be better to select a small compute cluster.
3. *VM size*: similar to the cluster size, increasing the amount of RAM, number of cores, and processing speed of the VMs will reduce the computational time, but it will be more expensive.
4. *Dedicated versus low-priority resources*: dedicated resources are non-interruptible, while low-priority instances can be assigned by Azure to other tasks and interrupt the job.

For this project, we will select a VM with a CPU, and from the listed VMs we can select the one with optimized memory, which costs $0.32 per hour.

There are several optional steps that we can skip and click on `Review + Create` to create the compute instance.

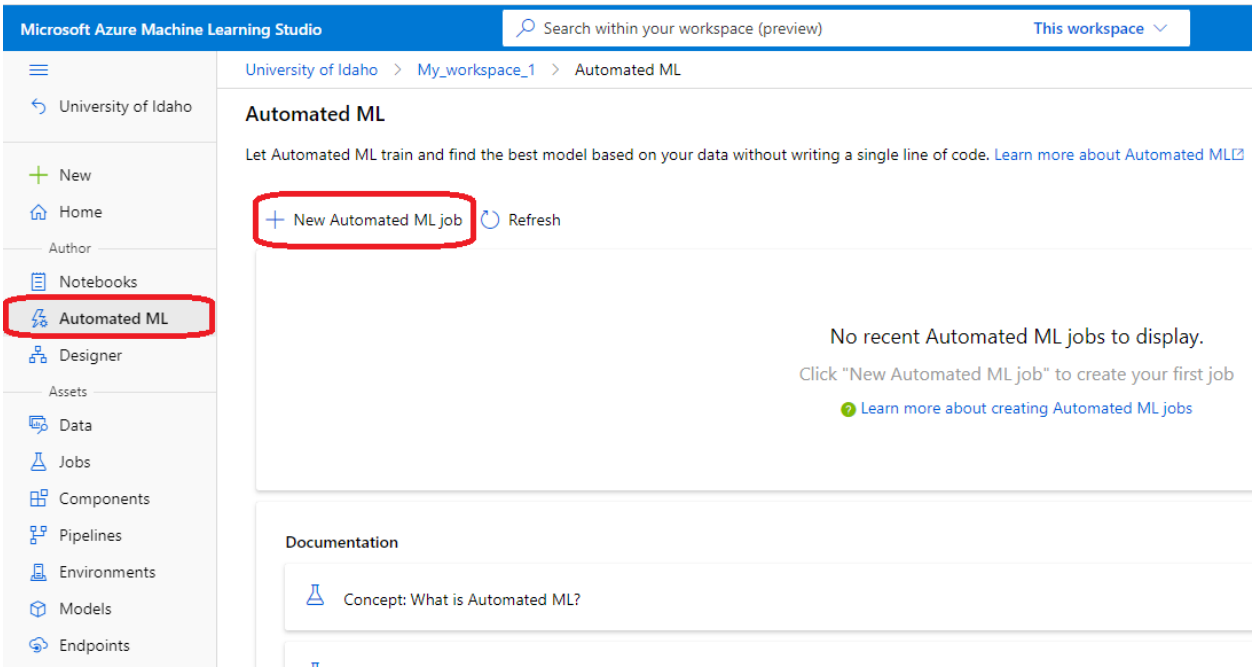
It can take a few minutes for the compute resource to be created.



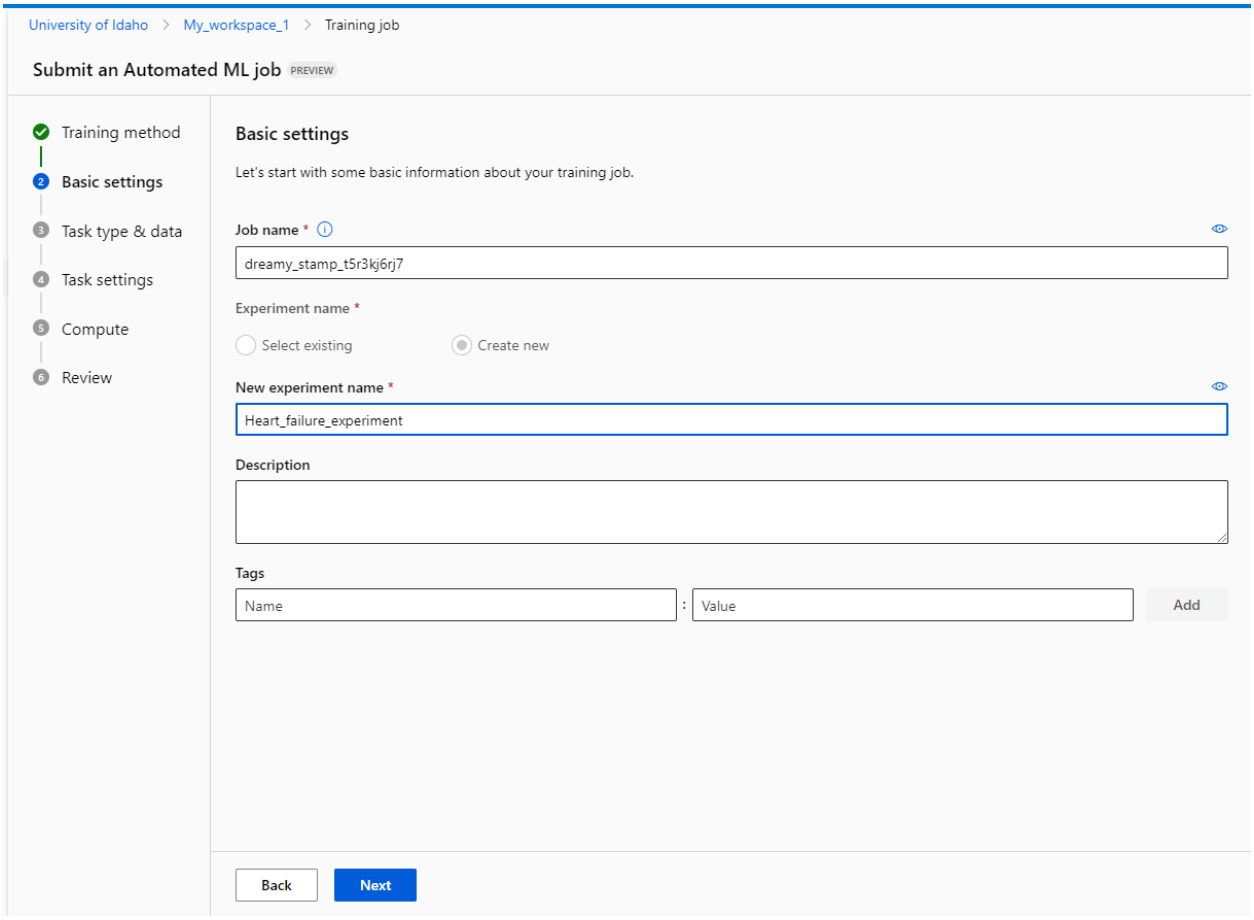
26.3.5 Training a Model with Auto ML

AutoML in Azure ML Studio allows to build and deploy ML models without writing code.

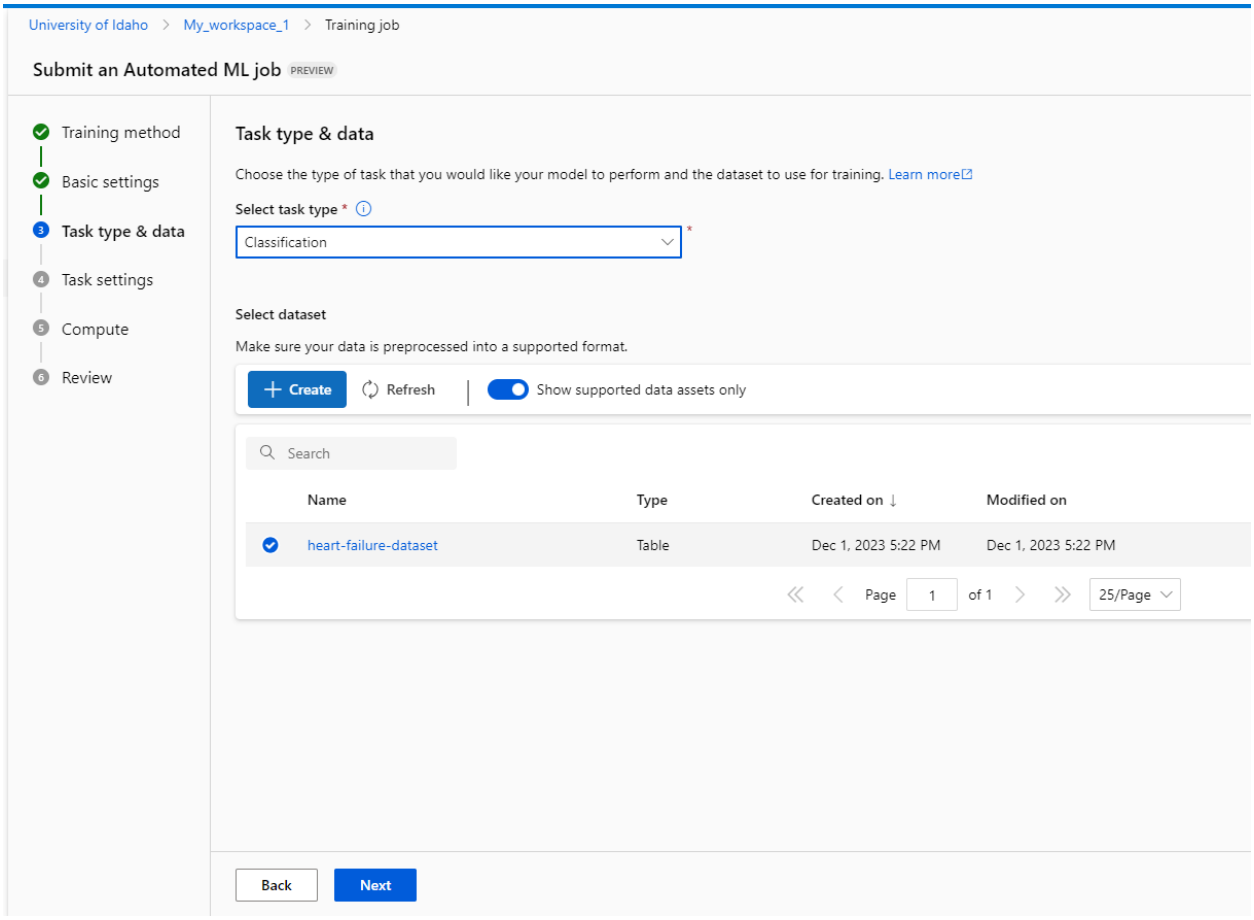
- Select `Automated ML` from the modules in the left-side panel.
- Select `+ New Automated ML job`.



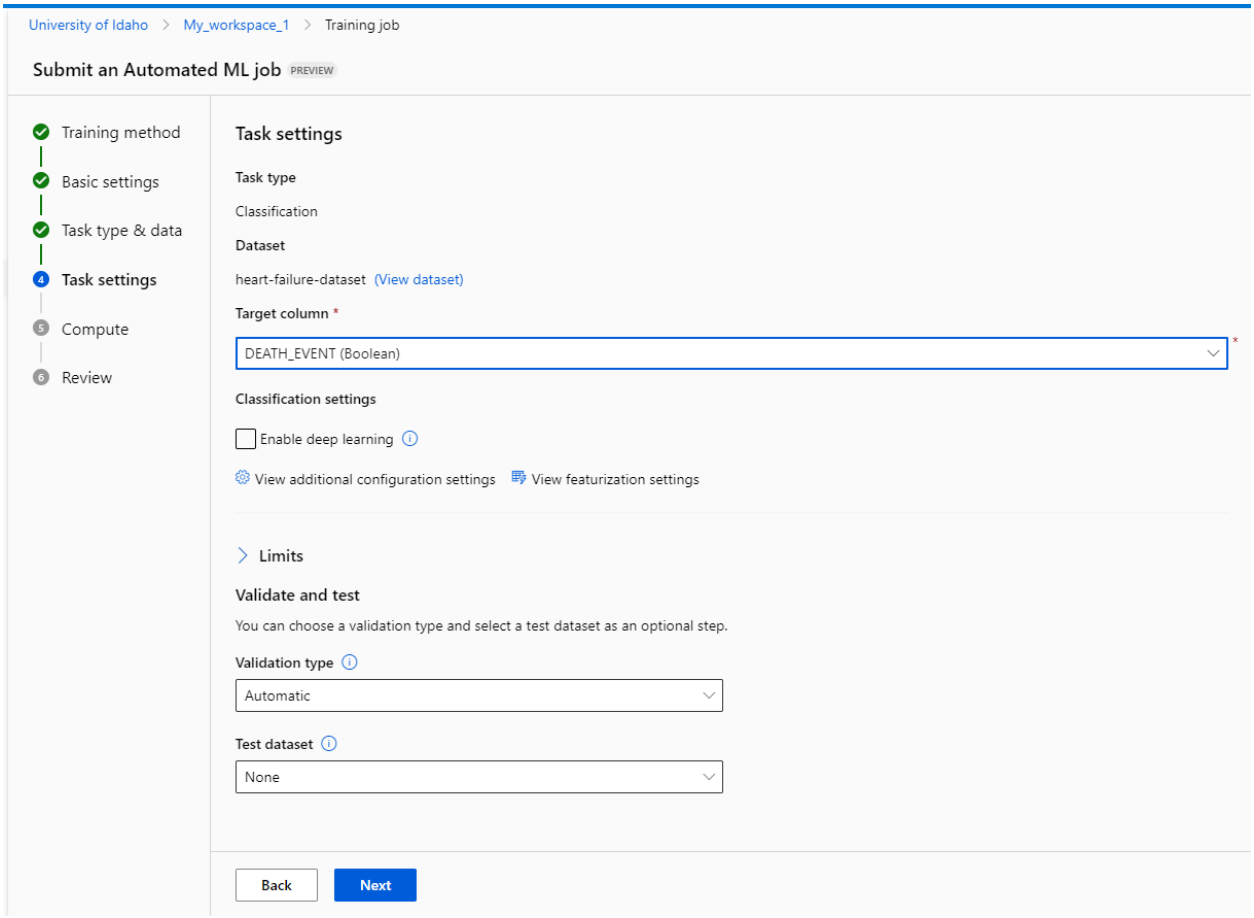
- The first step requires to assign a name for the experiment. Let's simply name it `Heart_failure_experiment`.



- Afterward, we need to indicate the *task*, that is, whether the goal is to perform classification, regression, time-series forecasting, etc. In this case, we select `classification`.
- For dataset, we will select the `heart-failure-dataset` that we uploaded.



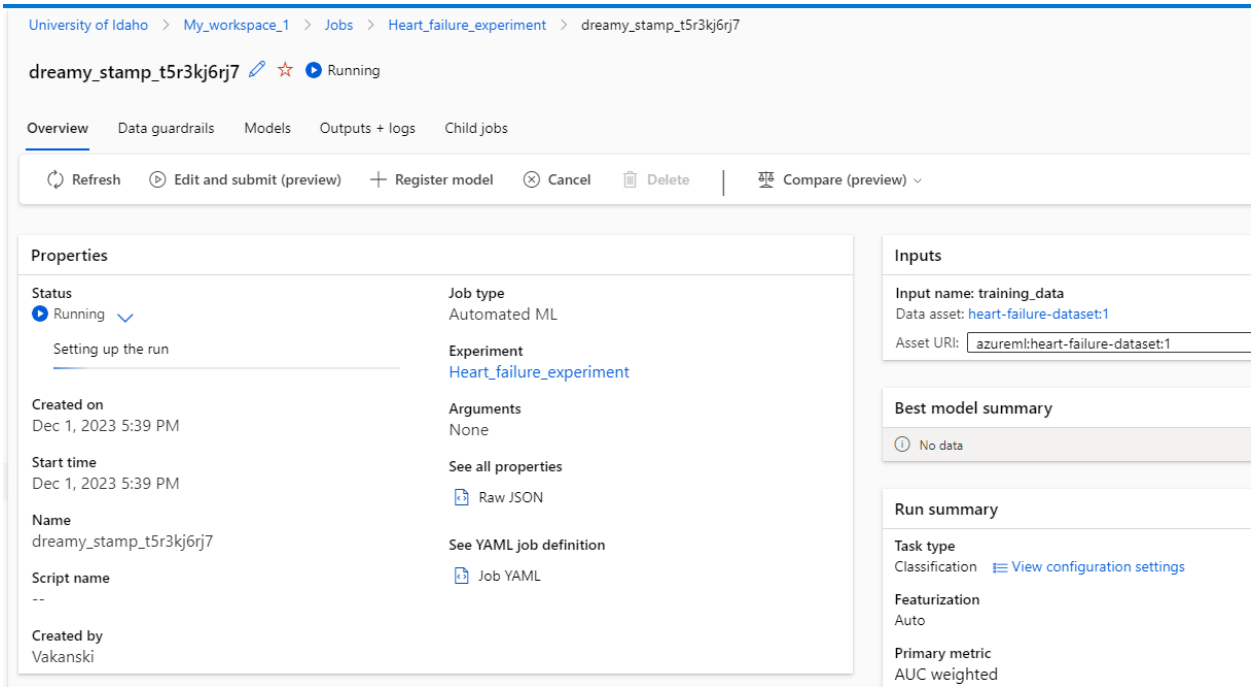
- Next, select a target column in the data: in this case it is `DEATH_EVENT`.
- We can also specify the validation type, e.g., whether we would like to use k-fold cross-validation, or whether to split the training data into train and validation sets, etc. Also, the test data asset field allows to upload a test dataset or specify how to evaluate the model. We can leave these fields at their defaults.



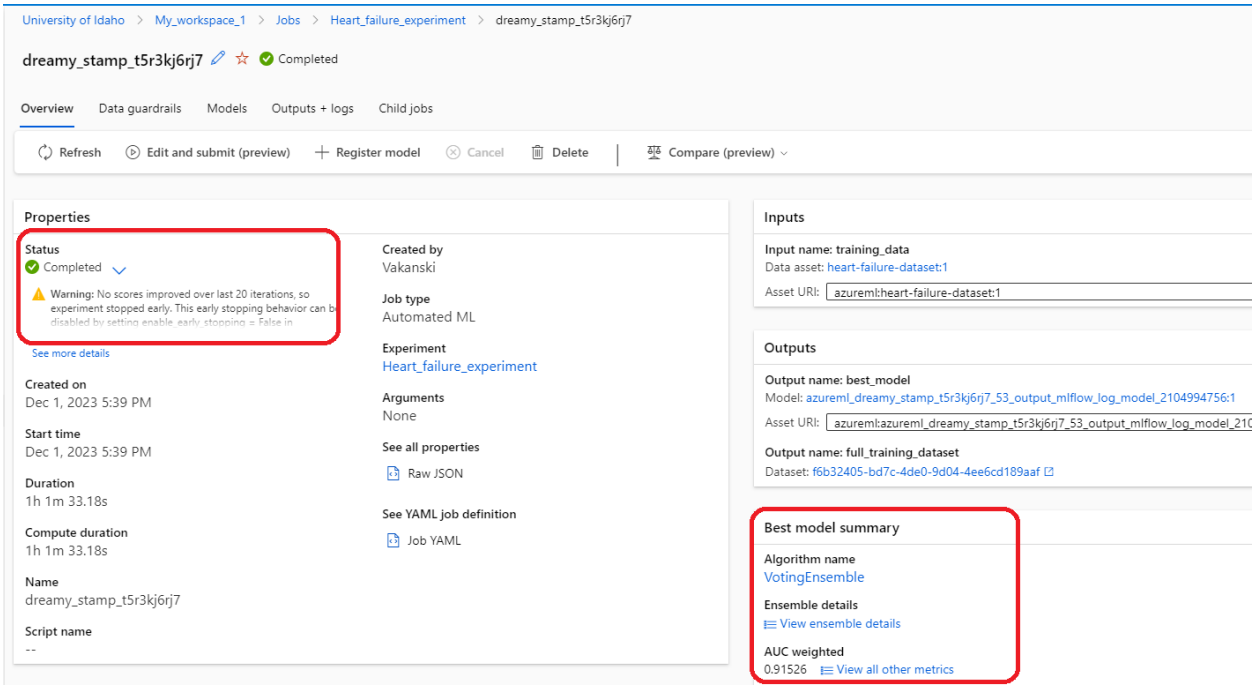
- Finally, use the drop-down menu to assign a compute resource, e.g., select the `heart-failure-compute` that we created.
- Review the AutoML job and submit it.

Now the setup is complete, the experiment will begin running. This means that Azure ML will train many different models, and explore different hyperparameters for the models.

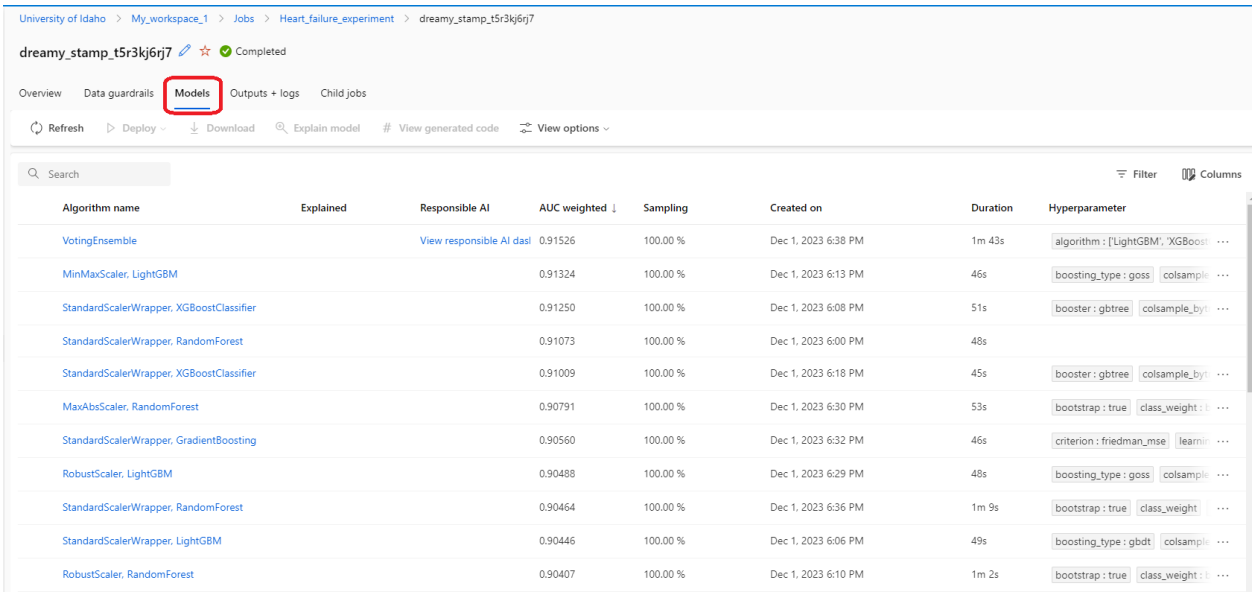
On the home page, under the `Jobs`section, we will see a summary of the entered information about the experiment, and we will also see that the status of the experiment is `Running`. It took about 1 hour to complete this experiment.



When the experiment is completed, in the `Best model summary` we can see that the highest performance was obtained by a Voting Ensemble model, which achieved 91.526% AUC.

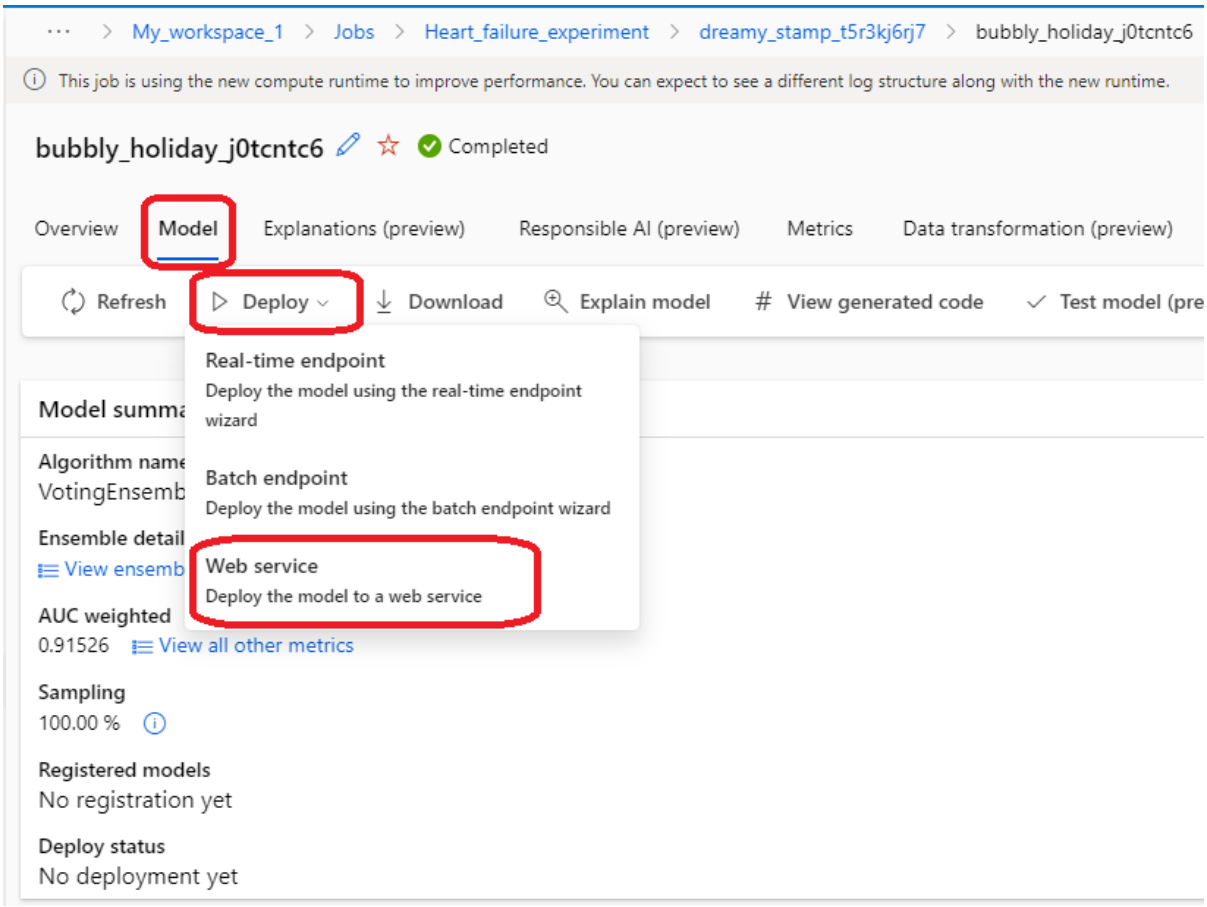


Also, let's select the `Models` tab to get more information about the training. We can see that over 50 models were trained in total, including LightGBM, XGBoost, Random Forest, Gradient Boosting, and running most of the models took under 1 minute. We can also see that different scaling methods were used with different algorithms (MinMaxScaler, RobustScaler, StandardScaler).



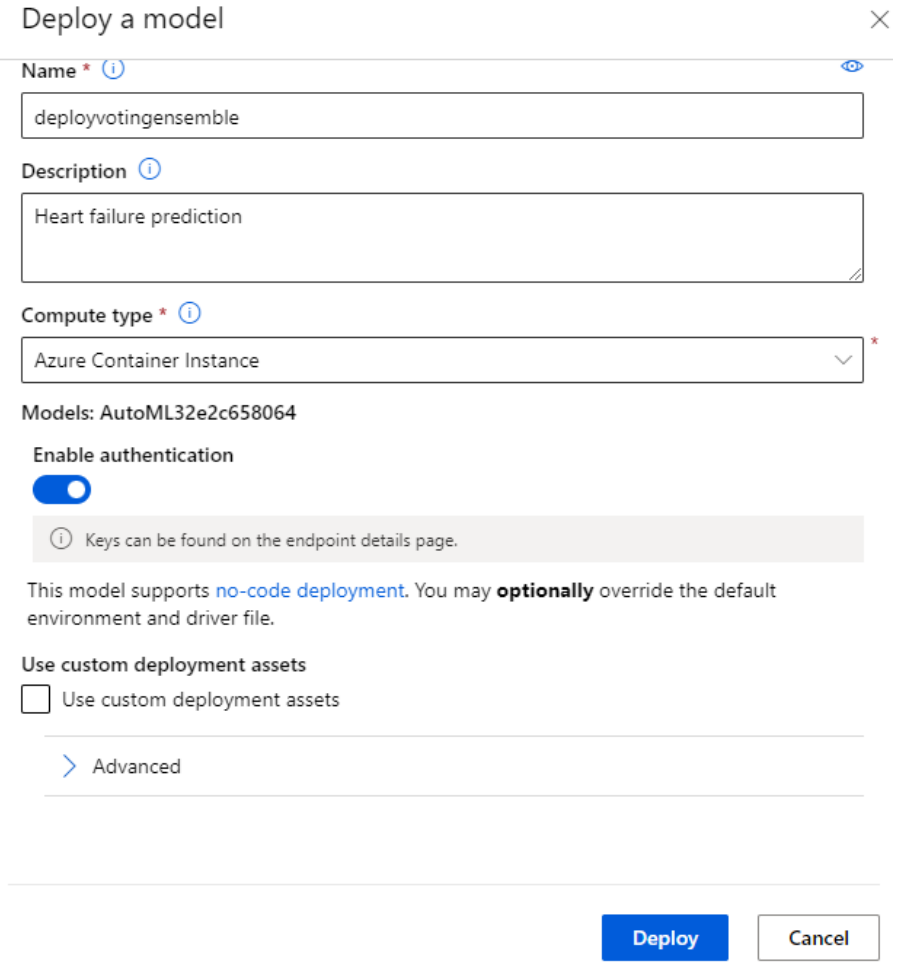
26.3.6 Deploying the Model

To deploy a trained model as a web service, we will select the Voting Ensemble as the best model, and from the drop-down menu under the `Deploy` tab select `Web service`.



In the newly opened form, we need to assign a name for the deployed model, a brief description, and compute type for the deployed model. In this case, we selected Azure Container Instance, which is suitable for low-scale CPU-based workloads, as is the model for this project. Deploying models that require large computational resources can require using other compute type with GPUs, larger RAM memory, or larger number of cores.

Next, let's click on `Deploy` to initialize this step. It took about 15 minutes for this project to be deployed. When it is completed, the `Deploy Status` on the dashboard will change from Running to Succeeded.

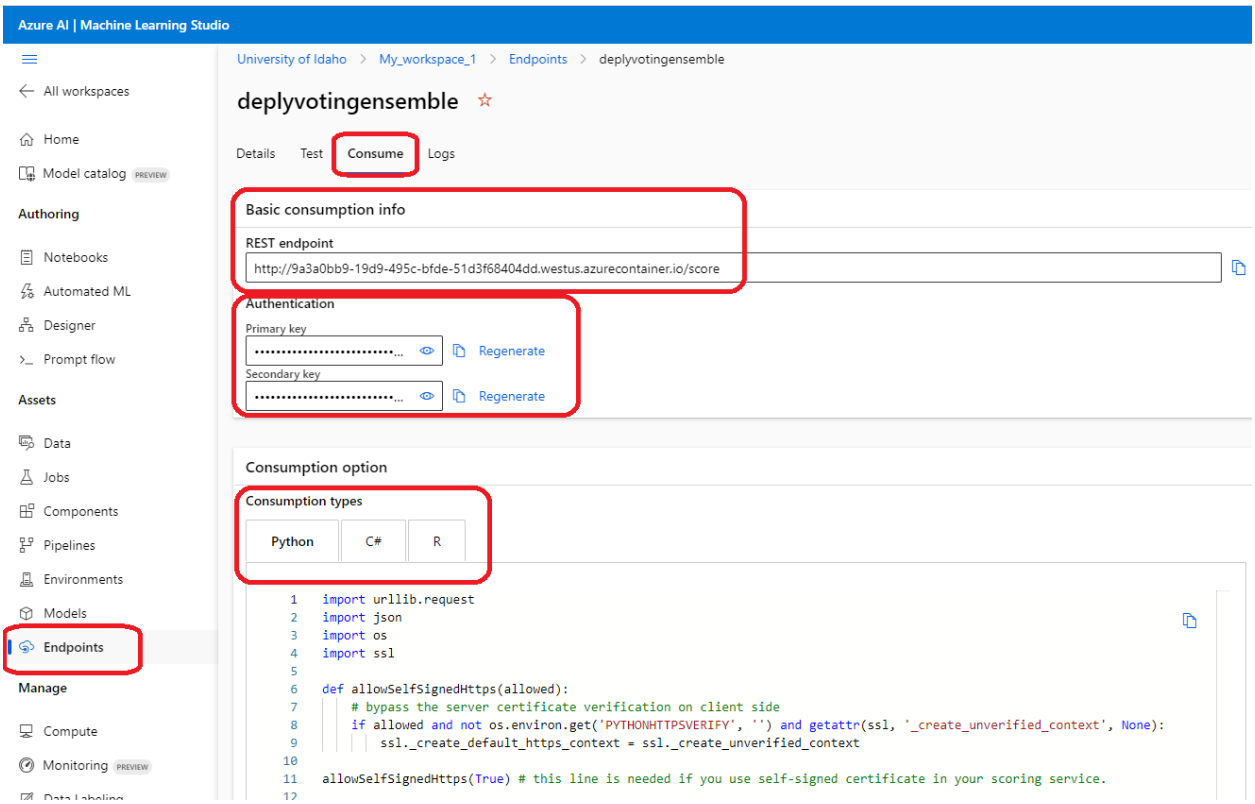


26.3.7 Consuming the Model

After the model is deployed, we can find the summarized information in the `Endpoints` module in the left-hand menu.

- Select the `Consume` tab to access the script for consuming the model.

The Consume page will provide the REST endpoint for users' consumption, the primary and secondary API keys for authentication, and a script for consuming the model from a local machine. The script is available in Python, C#, and R.



The Python script is shown below. The `data` section represents a dictionary where the users enter information for the input features. In this script, all values are set either to 0 or False. Then, the `url` in the code below is the address for the REST endpoint from the above figure, and `api_key` is the primary authentication key that is listed in the above figure as well. The last code section makes a prediction for the `DEATH_EVENT`, and the result is displayed.

```
# Note: the codes in this lecture are not required for quizzes or assignments

import urllib.request
import json
import os
import ssl

def allowSelfSignedHttps(allowed):
    # bypass the server certificate verification on client side
    if allowed and not os.environ.get('PYTHONHTTPSVERIFY', '') and getattr(ssl, '_create_
↪unverified_context', None):
        ssl._create_default_https_context = ssl._create_unverified_context

allowSelfSignedHttps(True) # this line is needed if you use self-signed certificate in␣
↪your scoring service.

# Request data goes here
# The example below assumes JSON formatting which may be updated
# depending on the format your endpoint expects.
# More information can be found here:
# https://docs.microsoft.com/azure/machine-learning/how-to-deploy-advanced-entry-script
data =  {
  "Inputs": {
```

(continues on next page)

(continued from previous page)

```
    "data": [
      {
        "age": 0.0,
        "anaemia": False,
        "creatinine_phosphokinase": 0,
        "diabetes": False,
        "ejection_fraction": 0,
        "high_blood_pressure": False,
        "platelets": 0.0,
        "serum_creatinine": 0.0,
        "serum_sodium": 0,
        "sex": False,
        "smoking": False,
        "time": 0
      }
    ]
  },
  "GlobalParameters": {
    "method": "predict"
  }
}

body = str.encode(json.dumps(data))

url = 'http://a836b469-9573-4a63-bd31-90e8205ae13c.westus2.azurecontainer.io/score'
api_key = 'QYEGDiZFpL5OECR2aZEhhNfSfYhPvgVn' # Replace this with the API key for the web␣
↪service

# The azureml-model-deployment header will force the request to go to a specific␣
↪deployment.
# Remove this header to have the request observe the endpoint traffic rules
headers = {'Content-Type':'application/json', 'Authorization':('Bearer '+ api_key)}

req = urllib.request.Request(url, body, headers)

try:
    response = urllib.request.urlopen(req)

    result = response.read()
    print(result)
except urllib.error.HTTPError as error:
    print("The request failed with status code: " + str(error.code))

    # Print the headers - they include the requert ID and the timestamp, which are␣
↪useful for debugging the failure
    print(error.info())
    print(error.read().decode("utf8", 'ignore'))
```

To consume the model, we just need to save the script to our local machine and execute it. The output is shown below. For this set of input parameters, the result for DEATH_EVENT is `True`.

```
PS C:\Users\Alex\Documents\Codes\My Folder 2022\Python for Data Science Course\Lectures\Theme 4
Model Deployment\Lecture 26 - Deploying Project to the Cloud> python heart_failure_autoML.py
b'{"Results": [true]}'
```

Let's check the model prediction for the last record in the dataset. The input features are shown below.

```
"age": 50.0,
"anaemia": False,
"creatinine_phosphokinase": 196,
"diabetes": False,
"ejection_fraction": 45,
"high_blood_pressure": False,
"platelets": 395000.0,
"serum_creatinine": 1.6,
"serum_sodium": 136,
"sex": True,
"smoking": True,
"time": 285
```

The prediction by the model is `False` as expected.

```
PS C:\Users\Alex\Documents\Codes\My Folder 2022\Python for Data Science Course\Lectures\Theme 4
Model Deployment\Lecture 26 - Deploying Project to python heart_failure_autoML_1.py
b'{"Results": [false]}'
```

7.26.4 26.4 Code-based Azure ML

In this section, we will use **Azure ML Studio** to manage Data Science projects in a Python environment, that can include Jupyter Lab, Jupyter Notebooks, or VS Code. Differently from the previous section which focused on No-Code environment with Azure ML Studio, this section focuses on Code-based environment with Azure ML Studio.

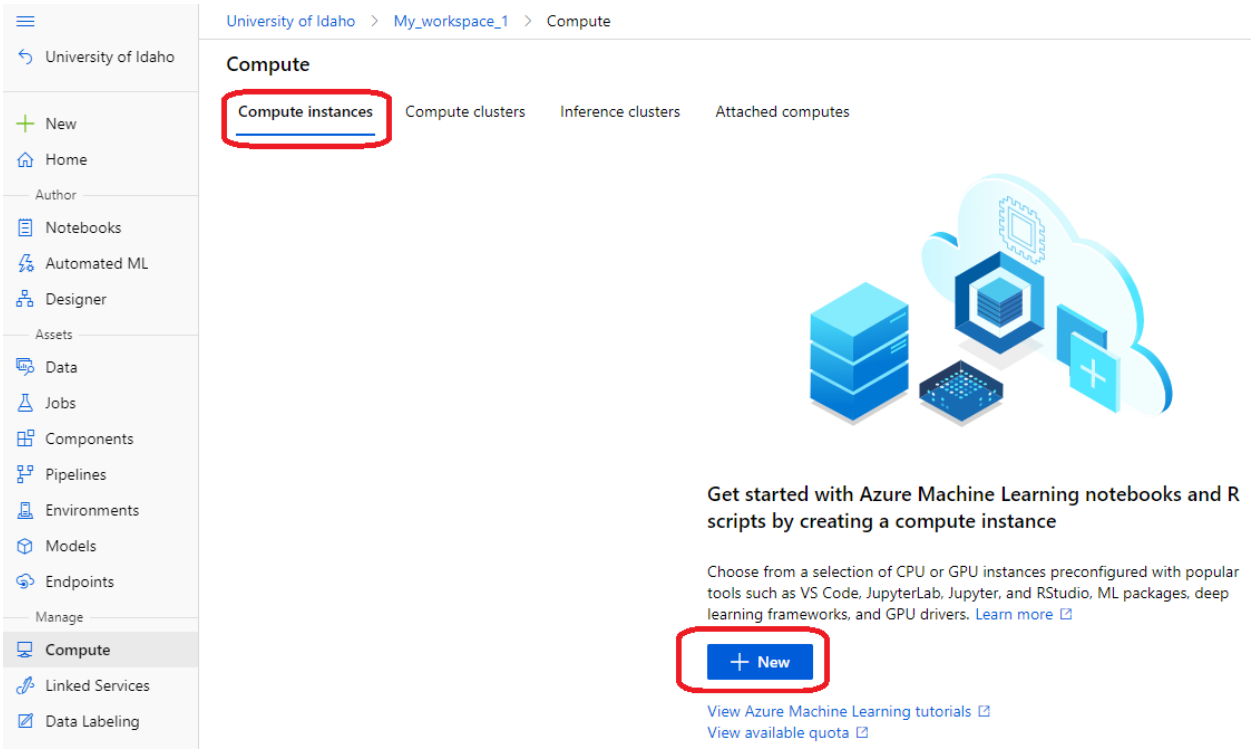
We will learn how to use Azure ML to train our own custom model. For this purpose, we will define a deep learning model for classification of the MNIST dataset, and we will train and evaluate the model using Azure ML resources. Afterward, we will deploy the model, and test the deployment.

If we didn't have access to GPUs from other sources, we could use the GPUs provided by Azure ML to train our models.

26.4.1 Creating Workspace and Compute Resource

Let's log in to the Microsoft Azure webpage and create a new workspace named `Workspace_2`, by following the steps listed in Section 26.3.

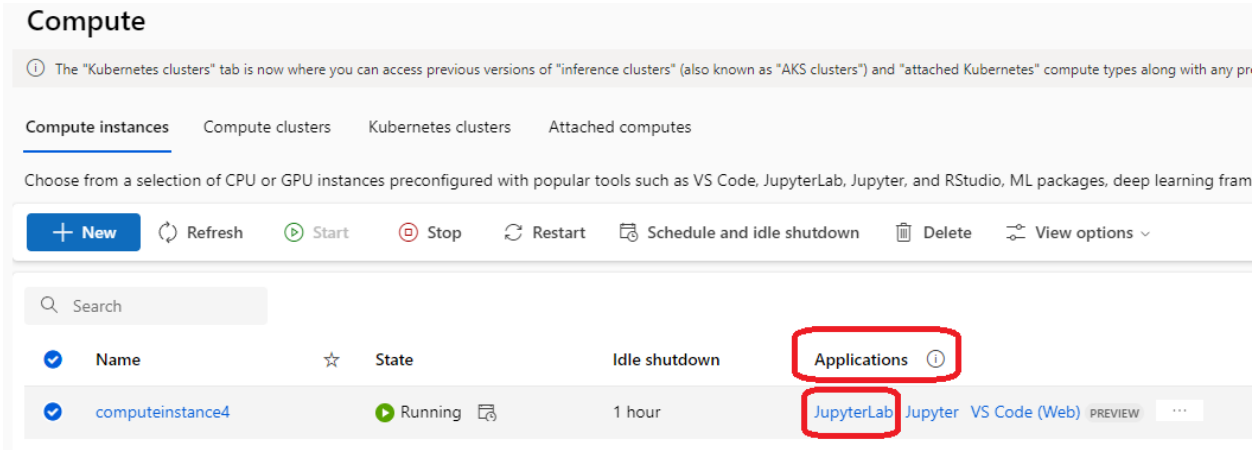
Afterward, we will navigate to the Azure ML Studio webpage, and in the newly created `Workspace_2`, we will create a new compute resource from the `Compute` module in the left-side menu. Similar to the previous section, we will select the `Compute instances` tab and click on `+ New`. For this task, we can use a CPU VM since MNIST is a relatively small dataset. If we were to work with larger datasets and models, we would need to select a GPU VM.



26.4.2 Using Jupyter Notebooks in Azure ML

The created compute instance will be listed on our homepage. Note that in the Applications tab, the listed applications include `Jupyter Lab`, `Jupyter`, and `VS Code`. We can use these applications to work with Jupyter Notebook files in the same way as we do outside of Azure ML Studio.

Let's select `Jupyter Lab` from the Applications tab for the newly created compute instance.



26.4.3 Loading the Data and Defining the Model

In the opened Jupyter Lab environment let's create a new notebook for training the model using `Python 3.8 - Pytorch and Tensorflow` kernel.

Let's rename the notebook to `mnist-demo`.

The code in the next cells is familiar, and it simply imports libraries and loads the MNIST dataset.

```
# Import libraries
import urllib.request
import tensorflow as tf
from tensorflow import keras
from keras.datasets import mnist
from keras.models import Model
from keras.layers import Input, Dense, Dropout, Flatten, Conv2D, MaxPooling2D
import numpy as np
import matplotlib.pyplot as plt
```

```
# Load the data
(X_train, y_train), (X_test, y_test) = mnist.load_data()

print('Data shape:', X_train.shape)
```

We will use TensorFlow-Keras library to define a simple Convolutional Neural Network for MNIST classification.

```
# Define the layers in the model
inputs = Input(shape=(28, 28, 1))
conv1a = Conv2D(filters=32, kernel_size=3, padding='same')(inputs)
conv1b = Conv2D(filters=64, kernel_size=3, padding='same')(conv1a)
pool1 = MaxPooling2D()(conv1b)
flat = Flatten()(pool1)
dense1 = Dense(1024, activation='relu')(flat)
dropout1 = Dropout(0.5)(dense1)
outputs = Dense(10, activation='softmax')(dropout1)

# Define the model with inputs and outputs
model = Model(inputs, outputs)
```

```
# Compile the model
model.compile(optimizer="adam", loss='sparse_categorical_crossentropy', metrics=[
↪"accuracy"])
```

26.4.4 Preparing Azure ML Experiment and Training the Model

Next, we will create an Azure ML experiment, and we will associate it with the current workspace and the subscription information. Hence, we will assign the workspace name to the current `Workspace_2` using the information about our `Subscription` and `Resource group`.

Afterward, we will instantiate a new experiment named `demo-mnist-training` that will utilize the created Azure ML workspace and resources to train and deploy the model.

```
from azureml.core import Workspace, Experiment

SUBSCRIPTION="...enter you 32-digit subscription number here..."
GROUP="My_resource_group_1"
WORKSPACE="Workspace_2"

# Create an instance of the Workspace class using the subscription information
ws = Workspace(
    subscription_id=SUBSCRIPTION,
    resource_group=GROUP,
    workspace_name=WORKSPACE,
)

# Create an Azure ML experiment within the workspace "ws"
experiment = Experiment(ws, "demo-mnist-training")
```

Azure ML allows integration of the `MLflow` framework for managing and tracking ML experiments.

In the next cell, we import `MLFLow` and we will use it to automatically log the loss, accuracy, and other parameters of the training progress for the TensorFlow model with the `autolog()` method.

```
import mlflow, mlflow.tensorflow

# Track the experiment and log the training progress
mlflow.set_tracking_uri(ws.get_mlflow_tracking_uri())
mlflow.start_run(experiment_id=experiment.id)
mlflow.tensorflow.autolog()
```

Next, we train the model for 5 epochs, and we can see that it achieved close to 99% train accuracy.

Afterward, we terminate the `MLFlow` run, and save the model in the current directory.

```
# Train the model
model.fit(X_train, y_train, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 29s 14ms/step - loss: 0.3304 - accuracy: 0.8929
Epoch 2/5
1875/1875 [==============================] - 27s 14ms/step - loss: 0.0741 - accuracy: 0.9793
Epoch 3/5
1875/1875 [==============================] - 26s 14ms/step - loss: 0.0517 - accuracy: 0.9854
Epoch 4/5
1875/1875 [==============================] - 26s 14ms/step - loss: 0.0412 - accuracy: 0.9882
Epoch 5/5
1875/1875 [==============================] - 27s 14ms/step - loss: 0.0357 - accuracy: 0.9895
```

```
# End the mlflow run
mlflow.end_run()

# Save the model
model.save('mnist-tf-model.h5')
```

26.4.5 Consuming the Model

To consume the model, first we will register the model with Azure ML so that it can be used for inference in the future. This will save the model under the name `mnist-tf-model` and it will register it to our workspace. The registered model can be accessed from the `Models` section under `Assets` in the left-side panel in Azure ML Studio.

```
# Register the model
from azureml.core.model import Model

registered_model = Model.register(
    workspace=ws,
    model_name='mnist-tf-model',
    model_path='mnist-tf-model.h5',
    model_framework=Model.Framework.TENSORFLOW,
    model_framework_version=tf.__version__)
```

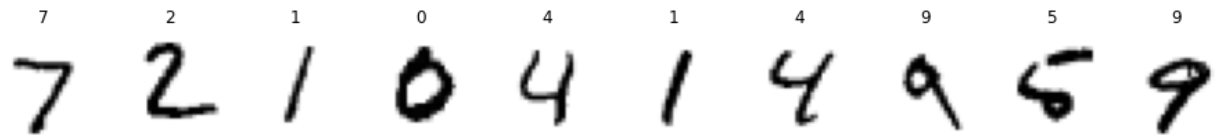
Afterward, we will load the registered model and use it to predict the classes for several images.

```
# load the registered model
aml_model = Model(workspace=ws, name='mnist-tf-model', version=registered_model.version)

downloaded_model_filename = aml_model.download(exist_ok=True)
print(downloaded_model_filename)

downloaded_model = tf.keras.models.load_model(downloaded_model_filename)
```

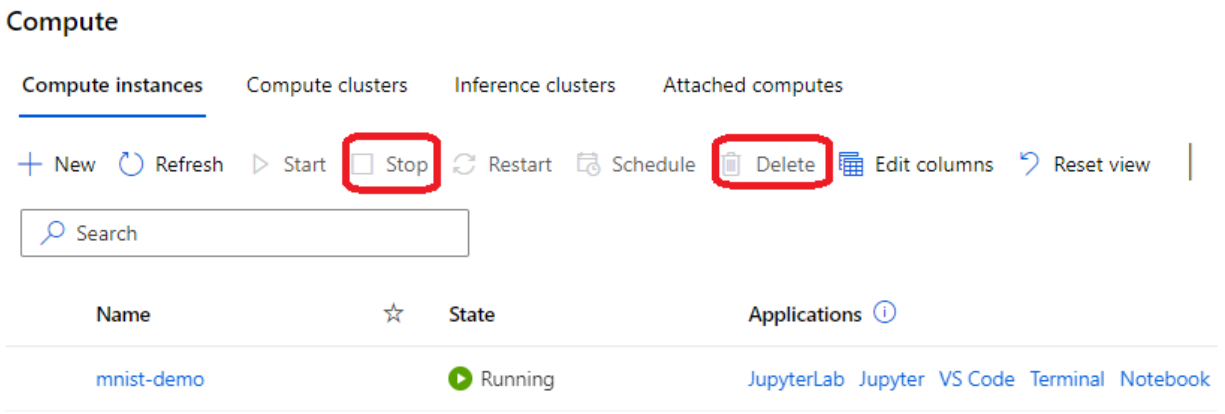
```
# Evaluate the model
downloaded_model.evaluate(X_test, y_test, verbose=0)
```

```
313/313 [==============================] - 2s 5ms/step - loss: 0.0311 - accuracy: 0.9910

[0.03105880692601204, 0.9909999966621399]
```

```
# Predict the labels for several images
preds = downloaded_model.predict(X_test).argmax(axis=1)
show_images(X_test[:10], preds[:10])
```



As we mentioned earlier, always remember to release the used compute resources after training or predicting with a model. One alternative is to `Stop` the current compute resource from running if we would like to reuse it later, or `Delete` the resources if they are not needed for future use.



7.26.5 References

1. Microsoft course - Data Science for Beginners, available at https://github.com/microsoft/Data-Science-For-Beginners.
2. From No-Code to Code in Azure Machine Learning, by William VanBuskirk, available at: https://levelup.gitconnected.com/from-no-code-to-code-in-azure-machine-learning-38ee6b556de2.
3. Creating a TensorFlow Model with Python and Azure ML Studio, by Jarek Szczegielniak, available at https://www.codeproject.com/Articles/5321728/Python-Machine-Learning-on-Azure-Part-3-Creating-a.

BACK TO TOP

7.27 Lecture 27 - Reproducible Data Science Projects

7.27.1 27.1 Introduction

Ensuring the reproducibility of Data Science projects is important for several reasons. One is that our customers (internal within your organization, or external) may use different operating systems, code dependencies, hardware, and they should be able to use our developed product and obtain the expected results. Another reason is collaborative development, where we may need to share our code with collaborators, and ensuring reproducibility of our codes requires information about the full environment in which the code was developed.

Predictions of Machine Learning models depend on the model architecture and parameters, code, libraries and dependencies needed to run the code. Ensuring reproducibility is a challenge that can affect the performance if not addressed. Regarding the model architecture and weights, we can simply save and load them, to ensure that the same model is used and that the predictions are consistent. Regarding code, we can use fixed random seeds and version control to ensure that the same data and code are used for training the model and making predictions. Dealing with code libraries and dependencies is more challenging, because libraries are updated frequently, and the model behavior can change, or even worse, the code can crash with some updates. To address this issue requires to apply a strategy for constraining the environment in which the model is developed and deployed.

The main strategies to constrain the libraries and dependencies include:

1. Interoperable standards,
2. Virtual environments,
3. Docker containers.

The first strategy involves standards like Open Neural Network Exchange (ONNX). ONNX is an open-source format for representing neural networks that allows interoperability between different frameworks (such as PyTorch and TensorFlow), or between different platforms. E.g., it allows a model trained in PyTorch to be imported and used for predictions in TensorFlow. Despite the potential, this strategy requires to frequently update the tools as the frameworks are updated, it has limited support for some operations in neural networks, it often has bugs in the conversions between the frameworks, and the performance of models can differ after the conversion to a target framework.

The second strategy is to use virtual environments in Python or Conda. Although using virtual environments can constrain the environment (e.g., Python dependencies), they provide only partial solution, since operating system-level dependencies or dependencies like CUDA that involve interaction with hardware may be outside the scope of virtual environments and cannot be addressed with this strategy.

The third strategy relies on using Docker containers and it is widely adopted at present time. **Docker** is an open-source library that uses containers to constrain project dependencies. Container is a standardized unit of fully packaged software used for local development, shipping code, and deploying systems.

7.27.2 27.2 Docker

An example of the reproducibility challenge with Data Science projects is shown in the next figure, depicting the multiple points for failure when we move our code to a cluster. These issues can include different versions of libraries, dependencies, drivers, or operating system. The main advantages of Docker containers are that they contain all dependencies, including hardware libraries, and ensure reproducible and consistent code performance.

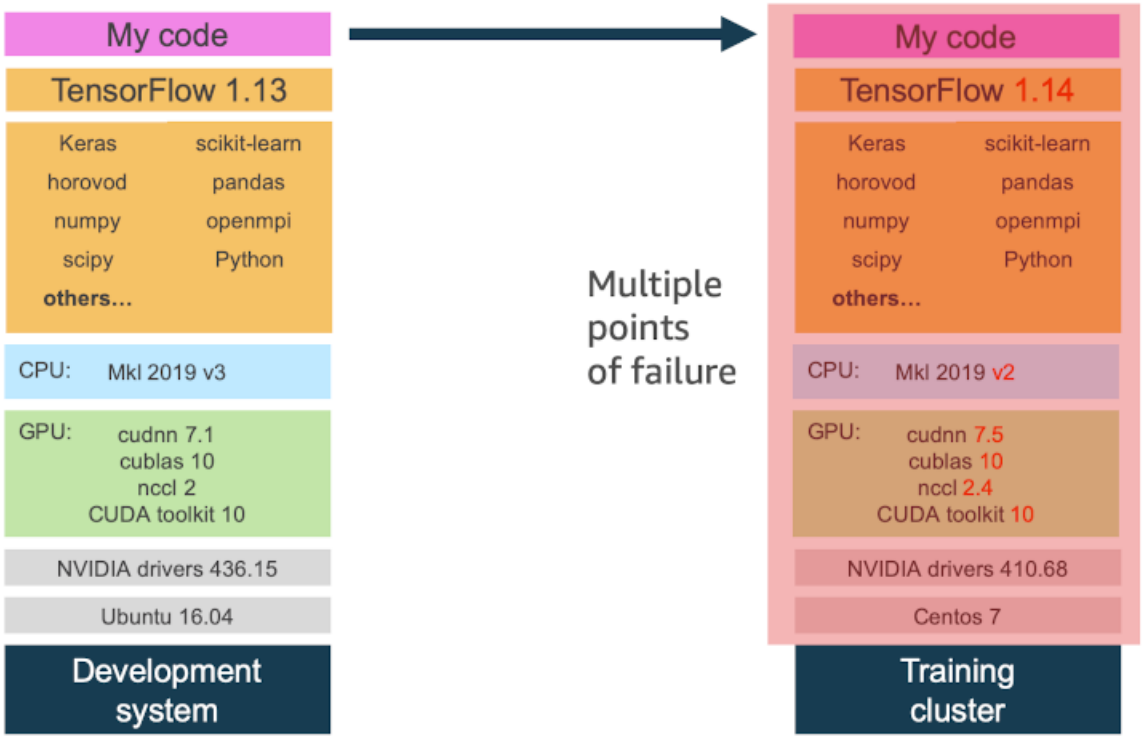


Figure: Reproducibility challenge.

Characteristics of containers are:

- Portable, since the containers are detached from the underlying hardware and the platform that runs them. This allows to seamlessly port our code and projects from our local machine to other machines (e.g., that may run on different operating system or use different hardware), or to external clusters that offer GPU resources. Portability enables teams to easily collaborate on projects and experiment with new software and frameworks, without the need to spend time setting up the environment for running code.
- Lightweight, since the containers don't require a full operating system, and they share the kernel of the host operating system.
- Scalable, applications can easily be scaled by adding or removing container instances.
- Secure, the application and its dependencies run in an isolated environment, hence the containers are less exposed to attacks.
- Facilitate deployment, where Data Science projects can be easily containerized and deployed for consumption by the end-users, with consistent performance between development and production environments.

Note also that there are notable differences between containers and Virtual Machines (VMs). VMs run a complete operating system with its own kernel, and each VM operates as an independent instance with its dedicated resources. Containers do not require hypervisor or hardware virtualization, and they use the host's operating system. This means that containers only need to package the application, its dependencies, and the minimal required components to run, without duplicating the entire operating system. Because of that, containers are smaller in size and faster to boot. VMs

offer stronger isolation than containers, because each VM operates independently of the host, whereas containers share the same kernel with the host. Orchestration tools for containers such as Kubernetes and Docker Swarm can be used to manage and scale containerized applications across clusters of machines, whereas orchestration of VMs is more complex. Containers are well-suited for microservices architectures, continuous integration and deployment (CI/CD), and scalable, distributed applications, while VMs are used for running legacy applications, and environments requiring strong isolation or using multiple operating systems.

Important concepts for working with Docker include:

- *Dockerfile* - script that defines the steps to build a Docker image.
- *Image* - is a built package containing the application and its dependencies.
- *Container* - is an instance where the Docker image is run.

We will explain these concepts through several examples in the following sections.

27.2.1 Installing Docker

Using Docker on a Windows machine can be done by the app Docker Desktop for Windows. This requires to download the app and install it. Depending on the Windows operating system, it may also require to install updates for Windows Subsystem for Linux, and other libraries. It is also possible to install Docker from the command line. Please follow the instructions on the Docker page for more detailed information.

7.27.3 27.3 Hello World in Docker

Let's start with a simple "Hello World" example in Docker, where the goal is to create a container with a Python file to print the Hello World! statement.

Step 1: Create a Python Script

For this purpose, we will create the Python script shown below that has just one line of code, and we will save it under the name `hello_world.py` in the folder `demo1`.

hello_world.py

```
1 print("Hello World!")
```

Dockerfile

```
1 FROM python:latest
2
3 COPY hello_world.py ./
4
5 CMD [ "python", "./hello_world.py"]
```

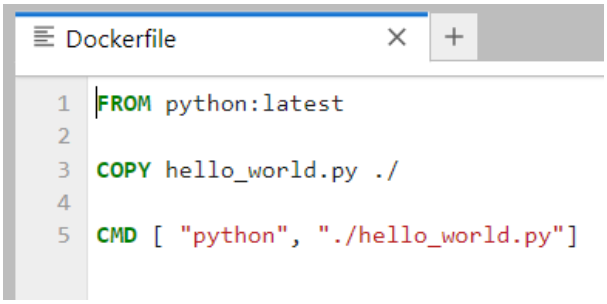
Step 2: Create a Dockerfile

Dockerfiles provide instructions for defining custom environments for our projects. I.e., for each particular project we will define the required software packages and libraries in a Dockerfile, and we will use the Dockerfile to build a Docker image.

It is common to start the Dockerfile with a *base environment*, which can simply include the latest Python version. Alternatively, as a base environment we can import pre-build environments by other developers. Docker provides access to *Docker Hub*, which is a registry with a large number of Docker images that have been published by other developers, and allows to directly import and take advantage of many pre-built environments.

Dockerfiles are plain text files, and for this simple project the Dockerfile is shown below:

- The first line uses the `FROM` command to instruct Docker to use Python as a base image for the environment. The term `latest` instructs Docker to retrieve the image of the latest official Python version. Docker will first check if we have the latest Python version installed on our machine, and if we don't, Docker will automatically download the image from Docker Hub. Alternatively, if we wanted to work with an earlier Python version, we can just write `FROM python:3.5` or `FROM python:2.7`, for example.
- The second line uses the `COPY` command to copy the created `hello_world.py` file from the local folder on my computer to the filesystem of the Docker image. The notation `./` is used for the root directory of the Docker image. The syntax is `COPY source_directory destination_directory`.
- The third line uses the `CMD` command to execute the `hello_world.py` script from the root directory in the image when the container is run.



The file needs to be saved under the name `Dockerfile`. The organization of the `Demo1` folder is:

```
Demo1
    ├── Dockerfile
    ├── hello_world.py
```

Step 3: Build the Docker Image

Docker image is a template that contains instructions for creating a container that runs on the Docker platform. It provides a convenient way to package up code and preconfigured environments, which we can use for our own private use or share publicly with other Docker users.

To build a Docker image from the Dockerfile, we will use the following code.

```
docker build -t demo1 .
```

Docker code begins with the keyword `docker`. The command `build` specifies to build a Docker image. `-t` is used to set a tag to the image, and in this case we tag the image with the name `demo1`. The dot `.` at the end of the line specifies to build the image by using the Dockerfile that is located in the current directory where `hello_world.py` script is located.

```
PS C:\Codes\My Folder 2022\Python for Data Science Course\Lectures\Theme 4 Model Deployment\Lecture 28 - Reproducible DS Projects\demo1> docker build -t demo1 .
[+] Building 0.6s (7/7) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 31B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/python:latest
 => [internal] load build context
 => => transferring context: 35B
 => [1/2] FROM docker.io/library/python:latest@sha256:10fc14aa6ae69f69e4c953cffd9b0964843d8c163950491d2138af891377bc1d
 => CACHED [2/2] COPY hello_world.py ./
 => exporting to image
 => => exporting layers
 => => writing image sha256:5cd86e402c131bd09bf5908023db6fc4975ac5cbfa6ed92285465130b8b8bf8b
 => => naming to docker.io/library/demo1
```

The Docker image that we just built represents a collection of files that are required for an operational environment. Each of the files that make up a Docker image is referred to as a *layer*. The images can contain multiple application codes and dependencies, i.e., they can have multiple layers.

In general, it is possible to build a Docker image by typing Docker commands interactively in the command line, and creating a Dockerfile is not required. However, Dockerfiles provide convenience and a documented record of the steps taken to assemble an image.

Step 4: Run the Docker Container

To run the image, we just use the `run` command, followed by the image tag `demo1`. This will execute the `hello-world.py` script in the container, which will display `Hello World!` in the terminal.

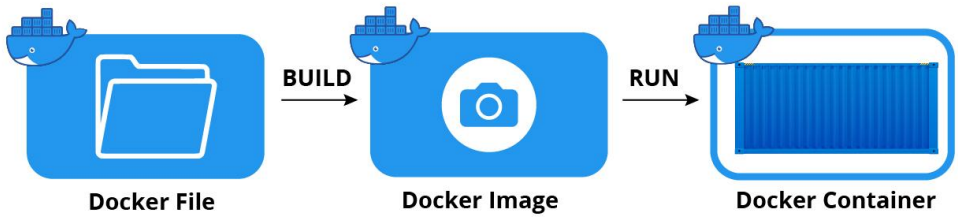
```
docker run demo1
```

```
PS C:\Codes\My Folder 2022\Python for Data Science Course\Lectures\Theme 4 Model Deployment\Lecture 28 - Reproducible DS Projects\demo1> docker run demo1
Hello World!
```

In summary, we created a container that is independent of our local machine and that has an implementation of Python, and we displayed a message in that container.

Conclusion

The figure below shows the basic steps in Docker. We begin with a Dockerfile that contains the required commands to build a custom Docker image for our application. The Docker image is a custom environment, that contains the required code, libraries, and dependencies for the application. Docker containers are run using the Docker image. Within a container, the image comprises all the required files to run the application. By using containers, applications are isolated from other processes, preventing other processes from affecting the current application.



Docker containers have an analogy with cargo containers used for shipping products. Once we place our product (code and dependencies) in the container, we can ship it by boat or train (laptop or cloud), and when it gets to its destination, it continues to function (run) just as before the shipment was made.

And one more note is that Docker has an official `Hello World` image, which can be run directly from the terminal, and it is typically used by users when they install Docker for the first time to verify that it works properly. The following figure displays the output of the `Hello World` image.

```
PS C:\Codes\My Folder 2022\Python for Data Science Course\Lectures\Theme 4 Model Deployment\Lecture 28 - Reproducible DS Projects\demo2-sklearn> docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:faa03e786c97f07ef34423fccceeec2398ec8a5759259f94d99078f264e9d7af
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

7.27.4 27.4 Scikit-learn Models in Docker

In this section, we will learn how to create a container for running a scikit-learn model.

To demonstrate the concept, we will use the *California Housing Dataset* that is available in scikit-learn, and that we also used in Lecture 15 when we studied regression with ANNs. The dataset contains 20,640 records about housing prices in California, and it includes 9 features related to the number of rooms, population, latitude, longitude, and similar information. The target column is the median value of the house. This is a regression problem, where the goal is for a given set of input features to predict the housing price.

For this task, we will create a new folder, containing a `Dockerfile` to build the Docker image, `train.py` script for loading the dataset, training the model, and saving the model parameters, and `inference.py` script for loading the trained model and predicting on test instances.

```
Demo2-sklearn
    ├── Dockerfile
    ├── train.py
    ├── inference.py
```

Step 1: Create `Train.py` script

The code in `train.py` script is shown below (note that if you run the cell in this Jupyter notebook, it will give errors, because the code is intended to be executed as a script).

This script is easy to understand. We import the necessary libraries, load the data, fit a Gradient Boosting Model, and save the model. From the imported `joblib` package we used the `dump` method to serialize the model parameters and save them at the `MODEL_PATH` location. Also note that the `train.py` script defines the paths to the directory `MODEL_DIR` and saved model `MODEL_FILE`. This will allow to pass the path to Docker at build time, and embed these locations into the Docker image. The `MODEL_DIR` and `MODEL_FILE` environmental variables are defined in the Doockerfile in the next section. Then, when the container is run, the script will read the locations of the files from the image, and use the saved model to predict on new data during inference.

[ ]:
```python
# Import libraries
import numpy as np
from sklearn import ensemble
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
```

(continues on next page)

(continued from previous page)

```
import os
from joblib import dump

# Load and split data
housing = fetch_california_housing(as_frame=True).frame
X = housing.drop('MedHouseVal', axis=1)
y = housing['MedHouseVal']
X_train, X_test, y_train, y_test = train_test_split(X,  y, random_state=13, test_size=0.
↪2)

# Fit regression model
model = ensemble.GradientBoostingRegressor()
model.fit(X_train, y_train)

# Directory paths
MODEL_DIR = os.environ["MODEL_DIR"]
MODEL_FILE = os.environ["MODEL_FILE"]
MODEL_PATH = os.path.join(MODEL_DIR, MODEL_FILE)

# Save the model
dump(model, MODEL_PATH)
```

Step 2 Create `Inference.py` script

The script `inference.py` is very similar to `train.py`, except that the trained model is loaded, and in the last few lines of code, the loaded model is used to predict the price for the first 10 instances from the test dataset. The output includes both the predicted prices and the ground-truth prices from `y_test`.

```
[ ]: # Import libraries
import numpy as np
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
import os
from joblib import load

# Load and split data
housing = fetch_california_housing(as_frame=True).frame
X = housing.drop('MedHouseVal', axis=1)
y = housing['MedHouseVal']
X_train, X_test, y_train, y_test = train_test_split(X,  y, random_state=13, test_size=0.
↪2)

# Directory paths
MODEL_DIR = os.environ["MODEL_DIR"]
MODEL_FILE = os.environ["MODEL_FILE"]
MODEL_PATH = os.path.join(MODEL_DIR, MODEL_FILE)

# Load model
model = load(MODEL_PATH)

# Run inference
```

(continues on next page)

(continued from previous page)

```
y_pred = model.predict(X_test[:10])
print("Predicted price by the model:", np.around(y_pred,1))
print("Ground-truth price:", np.array(y_test[:10]))
```

7.27.5 Step 3: Create `Dockerfile`

The `Dockerfile` for this project is provided below, and contains the following parts:

- `FROM jupyter/scipy-notebook` uses a pre-built image `jupyter/scipy-notebook` as a base image. The image contains a `python` installation, `scipy`, and other libraries to facilitate working with `Jupyter`. This image will be downloaded from Docker Hub.
- The next lines are used to first create a new directory called `model` where the trained model will be saved. Afterward, 2 environment variables are defined for `MODEL_DIR` and `MODEL_FILE` that will reference the newly created directory and the name of the saved model. The directory `/home/jovyan/` was set in the imported base image `jupyter/scipy-notebook`.
- The `COPY` commands are used to copy the `train.py` and `inference.py` scripts from our local machine to the working directory of the image.
- `RUN python train.py` will execute the script to train and save the model. This will ensure that the model is saved at a specific location , and it is ready to predict on new data when the image is run. An advantage is that if the model training fails, it will happen when we build the image rather than at run time, which allows to debug the issue.
- `CMD [ "python", "./inference.py"]` will execute the `inference.py` script when we run the container from the image.

[ ]:
```
FROM jupyter/scipy-notebook

RUN mkdir model
ENV MODEL_DIR=/home/jovyan/model
ENV MODEL_FILE=model.joblib

COPY train.py ./train.py
COPY inference.py ./inference.py

RUN python train.py

CMD [ "python", "./inference.py"]
```

Common commands that are used in Dockerfiles are shown in the next table.

Command	Purpose
FROM	To specify the base image.
WORKDIR	To set the working directory.
RUN	To install any libraries and packages required for the container.
COPY	To copy files or directories from a specific location.
ADD	To add remote URLs and unpack compressed files.
ENTRYPOINT	To specify commands to be executed when the container starts.
CMD	Commands the container executes.
EXPOSE	To define the port to access the container application.
LABEL	To add metadata to the image.

Step 4: Build the Docker Image

Next, we will build a Docker image from the Dockerfile, similarly to the previous example. Often, tags in the form `image_name:tag_name` are used, as in this case, where we assigned the image name `demo2-sklearn` and we assigned the tag name `0.1`. The tag name allows to apply versioning to Docker images, where, for instance, we can assign tag name `0.2` to the next version of image, or `1.0` if significant updates have been applied. If the tag name is omitted, Docker will pull the most recent image version, which is identified by the tag `latest`.

```
docker build -t demo2-sklearn:0.1 .
```

To list the Docker images that we have built, we can use, well, `docker images` shown in the figure. Note that the image `demo1` has the tag name `latest`, and the image `demo2-sklearn` has the tag name `0.1`.

```
PS C:\Codes\My Folder 2022\Python for Data Science Course\Lectures\Theme 4 Model Deployment\Lecture 28 - Reproducible DS Projects\demo2-sklearn> docker images
REPOSITORY      TAG       IMAGE ID       CREATED             SIZE
demo2-sklearn   0.1       b96327c969df   About an hour ago   3.03GB
demo1           latest    5cd86e402c13   46 hours ago        932MB
```

Now that we have an image that contains a saved trained model, next, we will run a container to make predictions using the model.

Step 5: Run the Docker Image

We run the image by using `docker run` and the name of the image. This will load the saved model, make predictions, and display the predicted house prices for the first 10 records in the test dataset. For comparison, the target prices for the first 10 records are also displayed. In a real-world case, the model will make predictions on new data samples (e.g., the price for house information that may be inputted by a realtor).

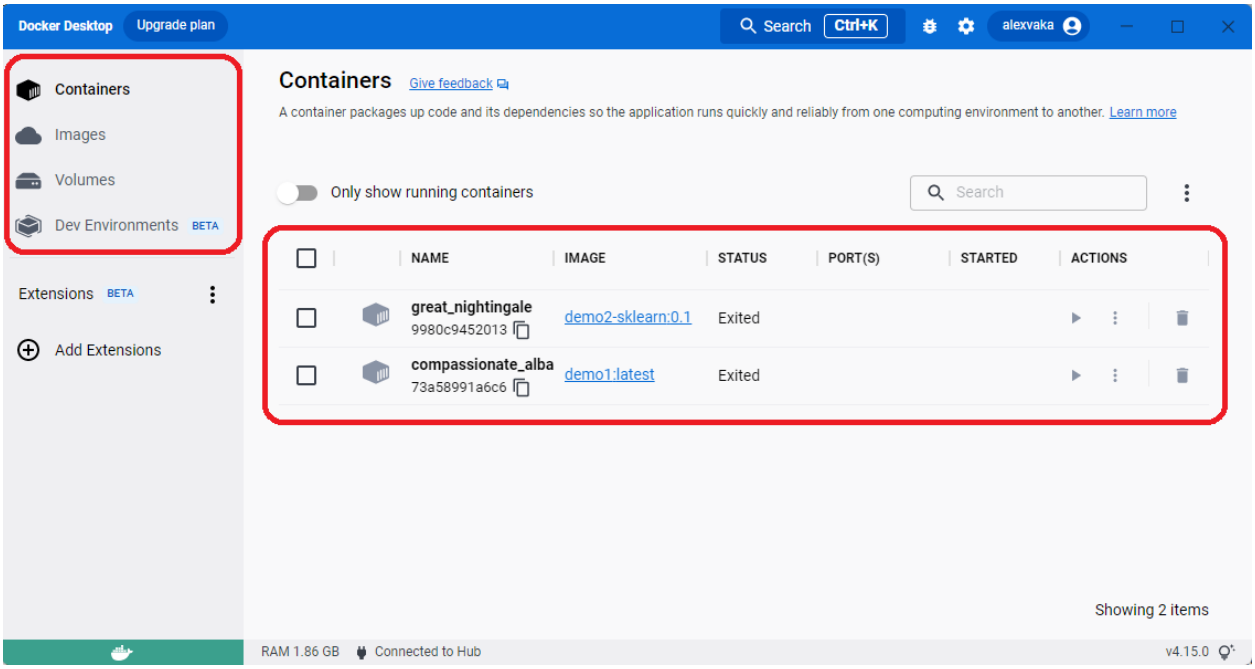
```
docker run demo2-sklearn:0.1
```

```
PS C:\Users\vakanski\Documents\Codes\2023 Codes\Python for Data Science Course\Lectures_2023\Theme_4-Model_Deployment\Lecture_27-Reproducible_Projects\demo2-sklearn> docker run demo2-sklearn:0.1
Predicted price by the model: [1.1 2.7 1.  0.6 1.5 2.5 2.3 1.6 2.6 2. ]
Ground-truth price: [1.3 2.3 0.8 0.5 1.1 2.7 2.8 1.4 2.2 1.9]
```

Note as well that we can launch multiple containers from one Docker image, with the containers maintaining their own individual state although sharing the same image. The changes to each individual container are stored in a *container layer*.

Docker Desktop

Docker Desktop for Windows also provides lists of all created images, containers, and other related information. For instance, in the Containers section, we can see the two containers that we run `demo1` and `demo2-sklearn` with tag names `latest` and `0.1`, respectively.



7.27.6 27.5 TensorFlow-Keras Models in Docker

One more similar example follows, which uses TensforFlow-Keras libraries for creating a Conv Net model for predicting the items in the Fashion MNIST dataset.

The organization of the folder is as follows:

```
Demo3-mnist
    ├── Dockerfile
    ├── mnist_train.py
    ├── mnist_inference.py
    ├── requirements.py
```

Step 1: Create `mnist_train.py` script

The script contains code for creating a Convolutional Neural Network model, compiling and training the model, and saving the model in a `cnn_model.keras` file.

[1]:
```
# Import modules
import tensorflow
from tensorflow import keras
from keras.models import Model
from keras.layers import Input, Dense, Dropout, Flatten, Conv2D, MaxPooling2D
```

(continues on next page)

(continued from previous page)

```
# Load the dataset
mnist_fashion = keras.datasets.fashion_mnist
(training_images, training_labels), (test_images,test_labels) = mnist_fashion.load_data()

# Scaling
training_images = training_images/255.0

# Define the layers in the model
inputs = Input(shape=(28, 28, 1))
conv1a = Conv2D(filters=32, kernel_size=3, padding='same')(inputs)
conv1b = Conv2D(filters=64, kernel_size=3, padding='same')(conv1a)
pool1 = MaxPooling2D()(conv1b)
flat = Flatten()(pool1)
dense1 = Dense(128, activation='relu')(flat)
dropout1 = Dropout(0.5)(dense1)
outputs = Dense(10, activation='softmax')(dropout1)
# Define the model with inputs and outputs
cnn_model = Model(inputs, outputs)

# Compile the model
cnn_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=[
↪'accuracy'])

# Fitting the model
cnn_model.fit(training_images, training_labels, epochs=5)

# Save the weights
cnn_model.save("cnn_model.keras")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-
↪labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 2us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-
↪images-idx3-ubyte.gz
26421880/26421880 [==============================] - 3s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-
↪labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0s/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-
↪images-idx3-ubyte.gz
4422102/4422102 [==============================] - 1s 0us/step
Epoch 1/5
1875/1875 [==============================] - 60s 32ms/step - loss: 0.4081 - accuracy: 0.
↪8565
Epoch 2/5
1875/1875 [==============================] - 60s 32ms/step - loss: 0.2771 - accuracy: 0.
↪8999
Epoch 3/5
1875/1875 [==============================] - 58s 31ms/step - loss: 0.2353 - accuracy: 0.
↪9140
Epoch 4/5
1875/1875 [==============================] - 61s 32ms/step - loss: 0.2074 - accuracy: 0.
↪9247
```

(continues on next page)

(continued from previous page)

```
Epoch 5/5
1875/1875 [==============================] - 57s 30ms/step - loss: 0.1841 - accuracy: 0.
↪9324
```

Step 2 Create `mnist_inference.py` script

The script `mnist_inference.py` is similar, where the saved model is loaded, and it is used to predict the digits in the first 10 images in the test dataset. For comparison, the real labels for the 10 test images are displayed as well.

[ ]:
```python
# Import modules
import numpy as np
import tensorflow
import tensorflow.keras as keras
from keras.models import load_model

# Load the dataset
mnist_fashion = keras.datasets.fashion_mnist
(training_images, training_labels), (test_images,test_labels) = mnist_fashion.load_data()

# Scaling
test_images = test_images/255.0

# loading the weights
cnn_model.load_model("cnn_model.keras")

# Predict
print("Predict the digit in the first 10 images in the test dataset")
y_pred = cnn_model.predict(test_images[:10])
print("Predicted image label:", np.argmax(y_pred,1))
print("Ground-truth image label:", test_labels[:10])
```

Step 3: Create `Dockerfile`

The `Dockerfile` is similar to the example in the previous section. It uses a Python 3.7.5 version, where `-slim` indicates to use lightweight image of Python. This will reduce the size of the image, but it may require to install additional libraries. Also note that the required libraries are provided in a `requirements.txt` file, instead of typing all required libraries individually in the Dockerfile. The rest of the lines copy the required files to the image, and then run the `requirements.txt` file to install the libraries in the image, and the `mnist_train.py` file to train the model and save it in the image.

```
numpy===1.21.5
keras===2.9.0
tensorflow===2.9.1
```

[ ]:
```
FROM python:3.7.5-slim

COPY requirements.txt ./requirements.txt
COPY mnist_train.py ./mnist_train.py
COPY mnist_inference.py ./mnist_inference.py
```

(continues on next page)

(continued from previous page)

```
RUN pip install -r requirements.txt
RUN python mnist_train.py

CMD [ "python", "./mnist_inference.py" ]
```

Step 4: Build and Run the Docker Image

Next, we will build the image `demo3-mnist` with a tag name `v1.0.0` and run it. The output is shown below.

```
docker image build -t demo3-mnist:v1.0.0 .

docker run demo3-mnist:v1.0.0
```

```
PS C:\Codes\My Folder 2022\Python for Data Science Course\Lectures\Theme 4 Model Deployment\Lecture 28 - Reproducible DS Projects\demo3-mnist> docker run demo3-mnist:v1.0.0
2022-12-06 23:23:08.390155: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot o
ectory
2022-12-06 23:23:08.390242: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.
2022-12-06 23:23:11.132983: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcuda.so.1'; dlerror: libcuda.so.1: cannot open shared
2022-12-06 23:23:11.133144: W tensorflow/stream_executor/cuda/cuda_driver.cc:269] failed call to cuInit: UNKNOWN ERROR (303)
2022-12-06 23:23:11.133234: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not appear to be running on this host (92dda9907f5b): /proc/driver/nvidia/
2022-12-06 23:23:11.134188: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the fol
ical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
Predict the digit in the first 10 images in the test dataset
1/1 [==============================] - 0s 181ms/step
Predicted image label: [9 2 1 1 6 1 4 6 5 7]
Ground-truth image label: [9 2 1 1 6 1 4 6 5 7]
```

7.27.7 27.6 Docker Registries and Repositories

Container repositories are the physical locations where Docker images are stored. Each repository contains different versions of each image, where each image is referenced by a different tag. For example, on Docker Hub, `mysql` is a repository that contains different versions of the Docker image for `MySQL`.

Container registry is a set of repositories. The main types of container registries are:

- Docker Hub: it is the official public container registry maintained by Docker, which hosts over 100,000 container images created by software vendors, open-source projects, and Docker's community of users.
- Third-party registry services: are managed registries that allow users to store and share their Docker images. Examples of third-party registries include Azure Container Registry, Google Container Registry, Amazon ECR, Red Hat Quay, and JFrog Container Registry.
- Self-hosted registries: are managed by organizations that prefer to host container images on their own on-premises infrastructure, typically due to security, compliance concerns, or lower latency requirements. Running self-hosted registries requires to deploy a registry server.

Pull and Push images on Docker Hub

As we mentioned, Docker Hub allows users to host and manage their own images, where Docker images can be easily shared between collaborators, allowing every collaborator to use reproducible environment.

To push built images to the Docker Hub registry, users will need to first log in to Docker Hub and then use the code to push the image.

```
docker login --username=your_username --password=your_password
docker push username/image:tag
```

Also, pulling images by other developers from Docker Hub is very easy, and allows to use other images with little or no modification.

```
docker pull username/image:tag
```

Benefits of Docker

- Code and dependencies can be packaged together into portable containers that can be run on a variety of different hosts, regardless of the hardware or operating system.
- Provides versioning of build environments, where image versions can be tracked and rolled back.
- Allow to replicate the working environment that is needed to train and deploy machine learning model on any system, and removes the reproducibility challenges.
- Trained machine learning models can be easily wrapped into an API in a container, and deployed to remote servers.

7.27.8 Appendix: Kubernetes

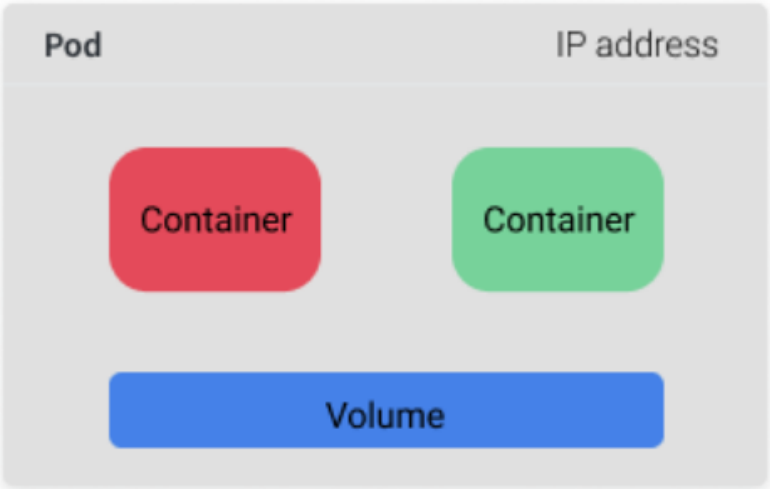
The material in the Appendix is not required for quizzes and assignments.

Although Docker containers provide the tools for ensuring reproducibility, when there are multiple containers that need to be run on a system consisting of multiple machines, managing the containers can become difficult. **Kubernetes** is a platform for orchestration of containers, that manages the coordination and communication between multiple containers. That is, for a set of containers and a set of available machines on a cluster, Kubernetes determines the optimal assignment of machines for each container. This is advantageous because it allows the system to function without the need for human intervention. E.g., if one container shuts down unexpectedly, it will run another container to replace its function.

The combination of Docker containers and Kubernetes orchestrator is well suited for developing and deploying *microservices*, where software applications are designed as a set of multiple, small, and decoupled services, that can be scaled and updated independently.

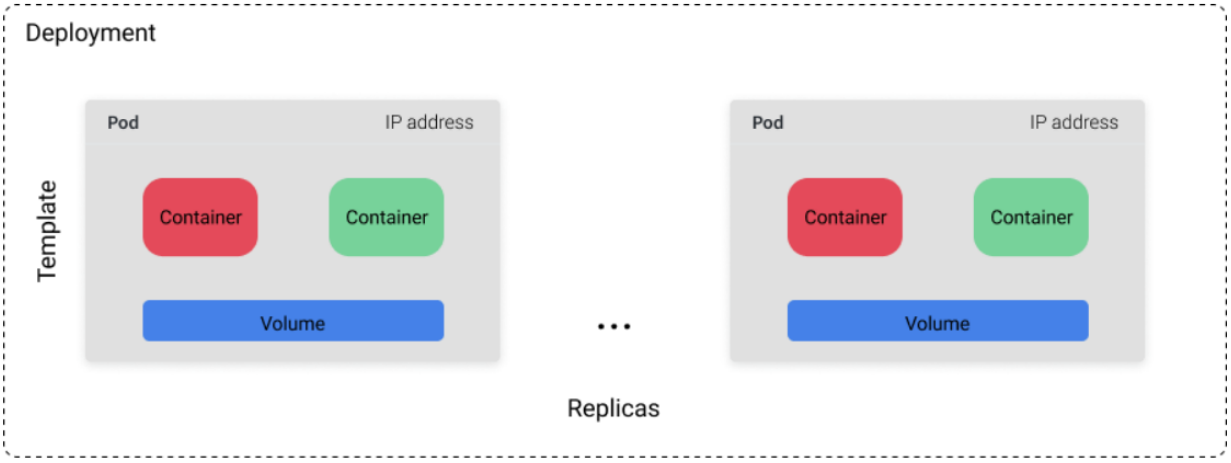
Pod

The building block of Kubernetes is a **pod**, consisting of one or multiple related containers, shared networking, and shared volume. The pods are designed as temporary objects, that will persist only for some period of time. Each pod is assigned a unique IP address to communicate with it. Volume is a storage location that is outside of a container, and allows to store data independently from the container. If a container is deleted, the volume is not deleted, and it can be used by other containers.



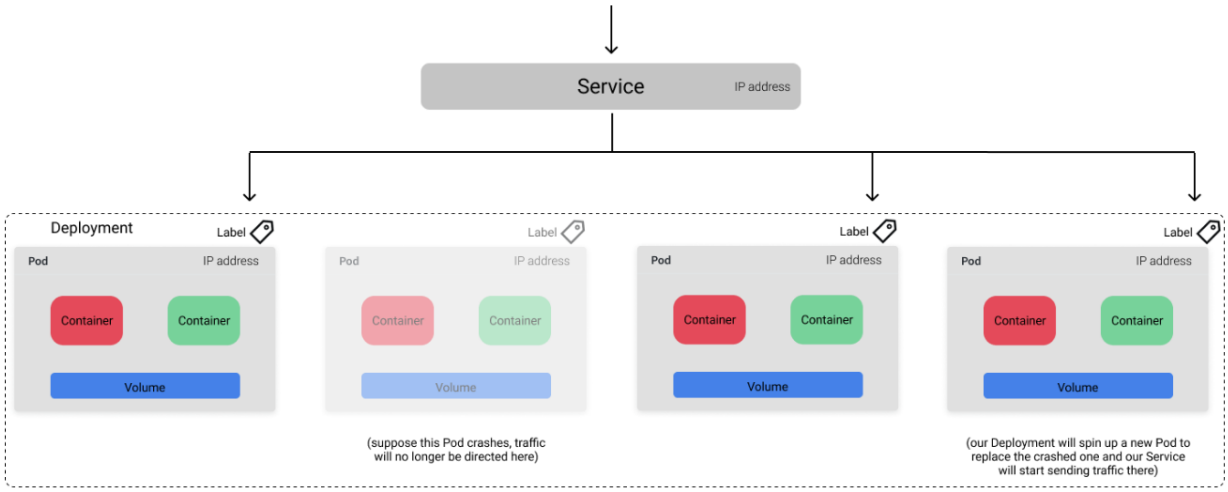
Deployment

Deployment object consists of several pods, which include a *template* pod and multiple *replicas* pods. Having replicas of pods allows to replace any crashing pods at any time.



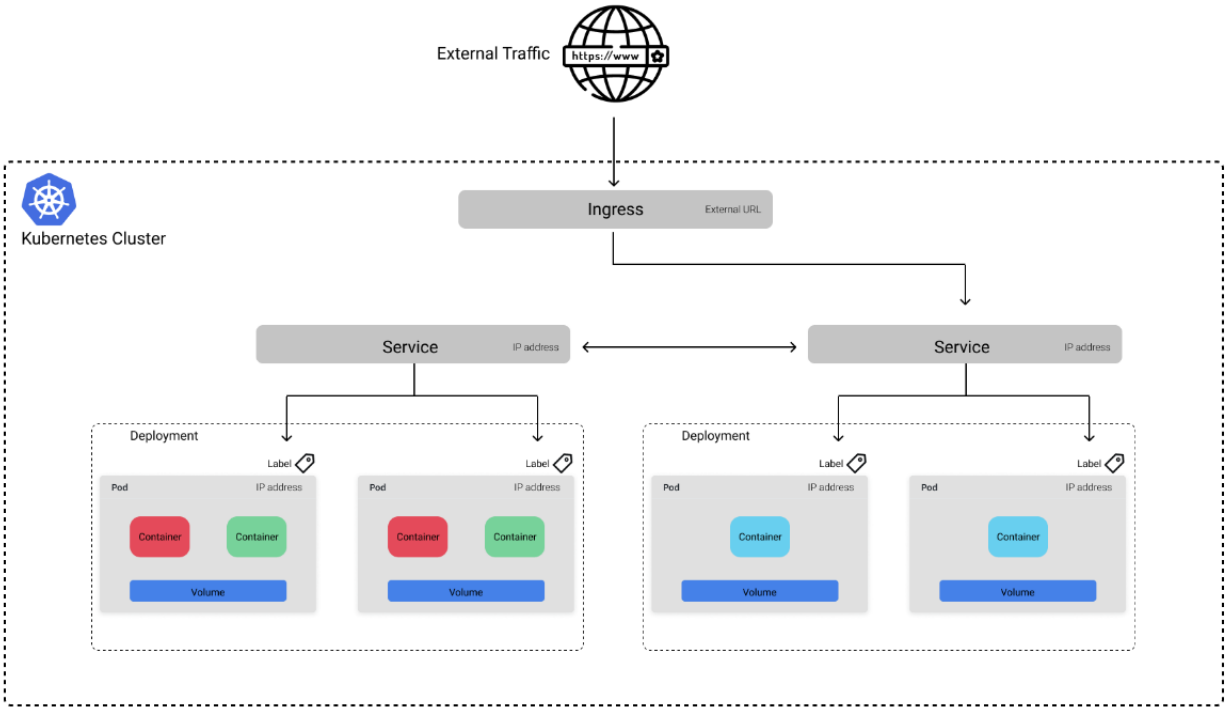
Service

Service object directs traffic to the pods, by balancing the incoming traffic between multiple pods. Also, if some pods crash, the service will launch new pods and it will direct the traffic to the new pods. Service also allows clients to access the pods without the need to know specific information about the pods (such as IP address or implementation).



Ingress

Ingress object controls which services that are exposed to external traffic. This allows to make certain services publicly available (e.g., predictions by a machine learning model) and keep certain services private (e.g., pods for training and updating the model with new data). In general, ingress refers to data entering a network, and egress is data leaving the network.



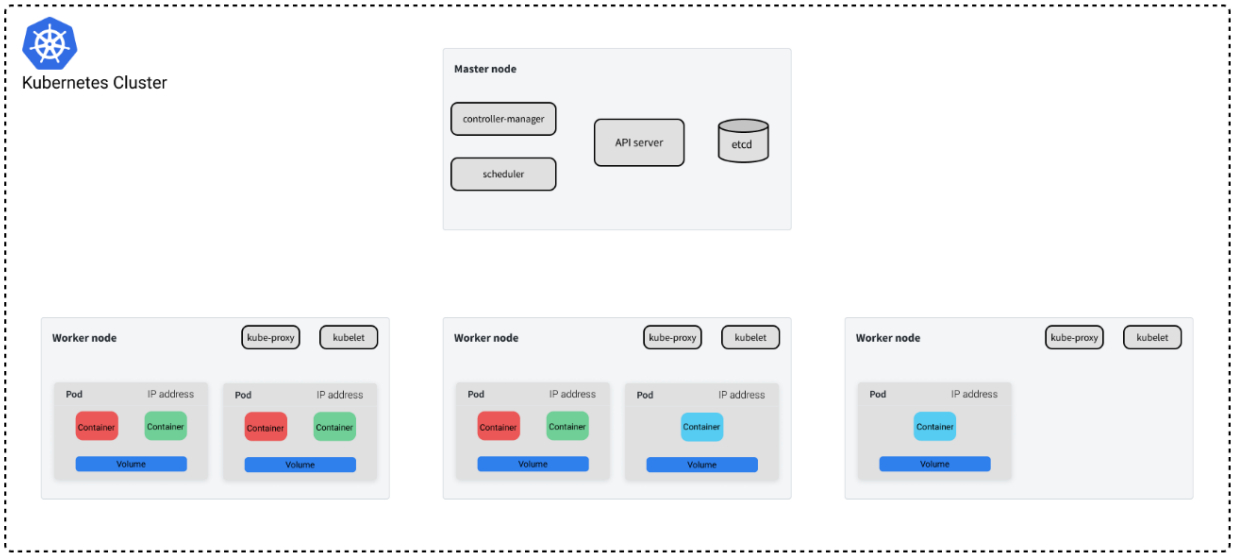
Control Plane

Kubernetes **control plane** controls how workloads are executed, monitored, and maintained on a cluster. A figure showing the elements in a control plane is shown below.

The cluster has a *master node* and multiple *worker nodes* that run the applications. For instance, a cluster can have several GPU-optimized worker nodes for model training and CPU-optimized worker nodes for making predictions.

The master node includes: scheduler - for assigning objects to available machines, controller manager - monitors the state of the cluster and compares the current state to the desired state, API server - communicates with a client and accepts requests, and etcd component - stores the current state of the cluster.

Each worked node includes a control component consisting of a kubelet and kube-proxy. The kubelet communicates with the API server to find out which workloads have been assigned to the node, and runs the pods to complete the assigned workloads. Kube-proxy enables containers from different pods to communicate with each other.



The Kubernetes system is designed to handle any failures in the cluster. For example, if the master node is down, the applications will not be immediately affected, and a new master node will be launched to continue the operation of the system.

Kubernetes are designed to orchestrate large number of workloads that may include multiple Data Science projects, distributed across clusters of servers, and serving continuously a large number of end-users. It is not the optimal solution for working with a single machine to run a lightweight load that does not need frequent changes and updates.

7.27.9 References

1. Full Stack Deep Learning course, Lecture 11: Deployment & Monitoring, available at https://fullstackdeeplearning.com/spring2021/lecture-11/.
2. ML in Production, Docker for Machine Learning, by Luigi Patruno, available at: https://mlinproduction.com/docker-for-ml-part-1/.
3. A Beginner's Guide to Understanding and Building Docker Images, available at: https://jfrog.com/knowledge-base/a-beginners-guide-to-understanding-and-building-docker-images/.
4. Machine Learning with Docker, Demonstrations about Using Docker for Machine Learning, available at: https://github.com/BINPIPE/docker-for-ml.
5. An introduction to Kubernetes, by Jeremy Jordan, available at: https://www.jeremyjordan.me/kubernetes/.

BACK TO TOP

7.28 Tutorial 1 - Jupyter Notebooks

7.28.1 Introduction to Jupyter Notebooks

Jupyter Notebook is an open-source web-based interactive computational environment, designed for sharing documents that contain code, text, equations, and visualizations. It is part of Project Jupyter, which also includes Jupyter Lab - an updated version of Jupyter Notebook with enhanced capabilities.

> **Jupyter notebooks** are documents that contain both code and text elements, such as equations, visualizations (i.e., figures and graphs), links, and similar related elements.

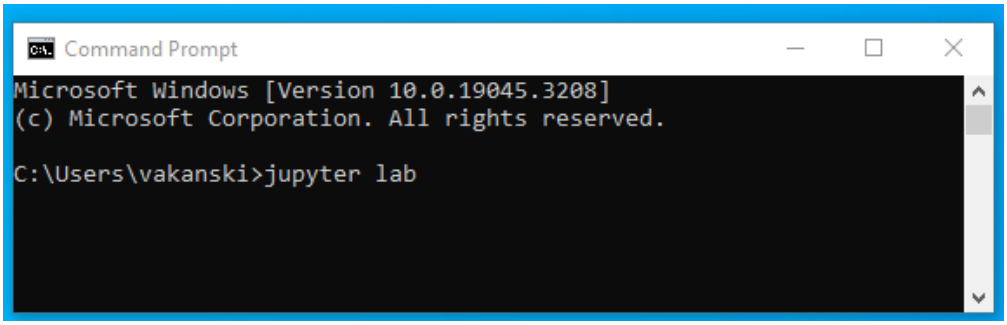
The name Jupyter does not really relate to the largest planet in the Solar system (spelled Jupiter), but instead, it is a coined word from the three core programming languages supported by Jupyter notebooks: Julia, Python, and R.

Jupyter Notebook was built upon an older IDE for Python called IPython.

The recommended user interface for this course is Jupyter Lab, since it offers advanced functionalities in comparison to Jupyter Notebook, such as access to a terminal, interactive widgets for exploring data, it allows multiple views into the same document, quick switching between opened files, etc.

Both Jupyter Lab and Jupyter Notebook are part of the Anaconda distribution, therefore if you have installed Anaconda on your computer, they will also be installed. And if you don't have Anaconda installed, you can install Jupyter Lab and Jupyter Notebook separately as any other package, e.g., `pip install jupyterlab`.

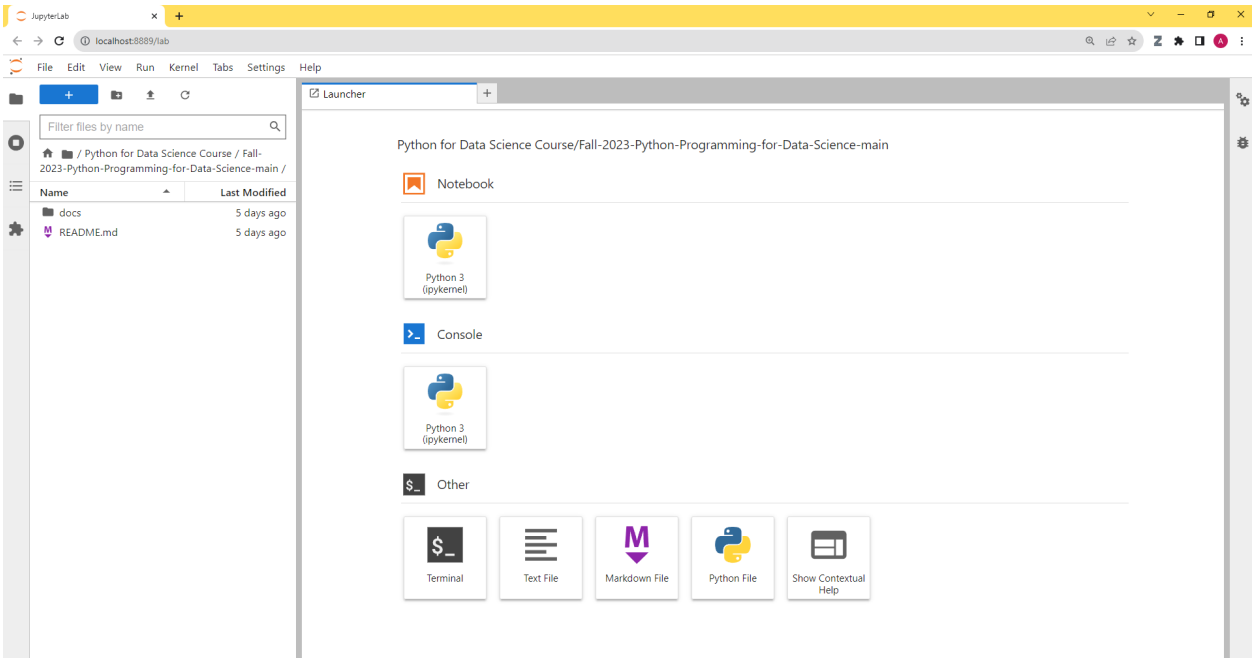
The Jupyter Lab environment can be started by running the command `jupyter lab` in the Command Prompt on Windows systems, or in a shell or terminal window on computers with other operating systems.



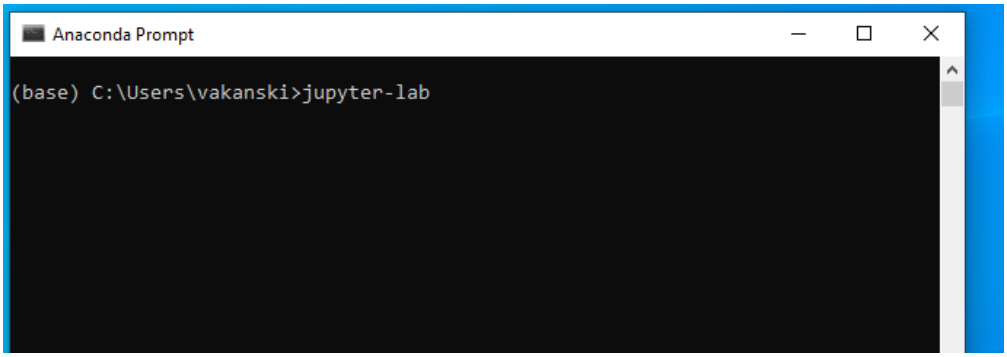
Jupyter will open in your default browser, or in a new tab if the browser has other open tabs, at the following URL: http://localhost:8889/lab. Localhost is not a website, it only indicates that the content is being served from your local machine. Therefore, Jupyter can be run on a computer without Internet access, or it can be run on a remote server accessed through the Internet.

Also, if you accidentally close the window, you can open it again while your server is running. For this example, navigating to http://localhost:8889/lab on your web browser will open it again.

The Jupyter Lab environment on my computer looks like in the figure below. The environment is also called Jupyter Dashboard, as it allows working with and managing notebooks. The start-up directory for the Jupyter Dashboard can be changed to a preferred directory.



It is also possible to open Jupyter Lab from the Anaconda Prompt, by typing `jupyter-lab` in the command line.

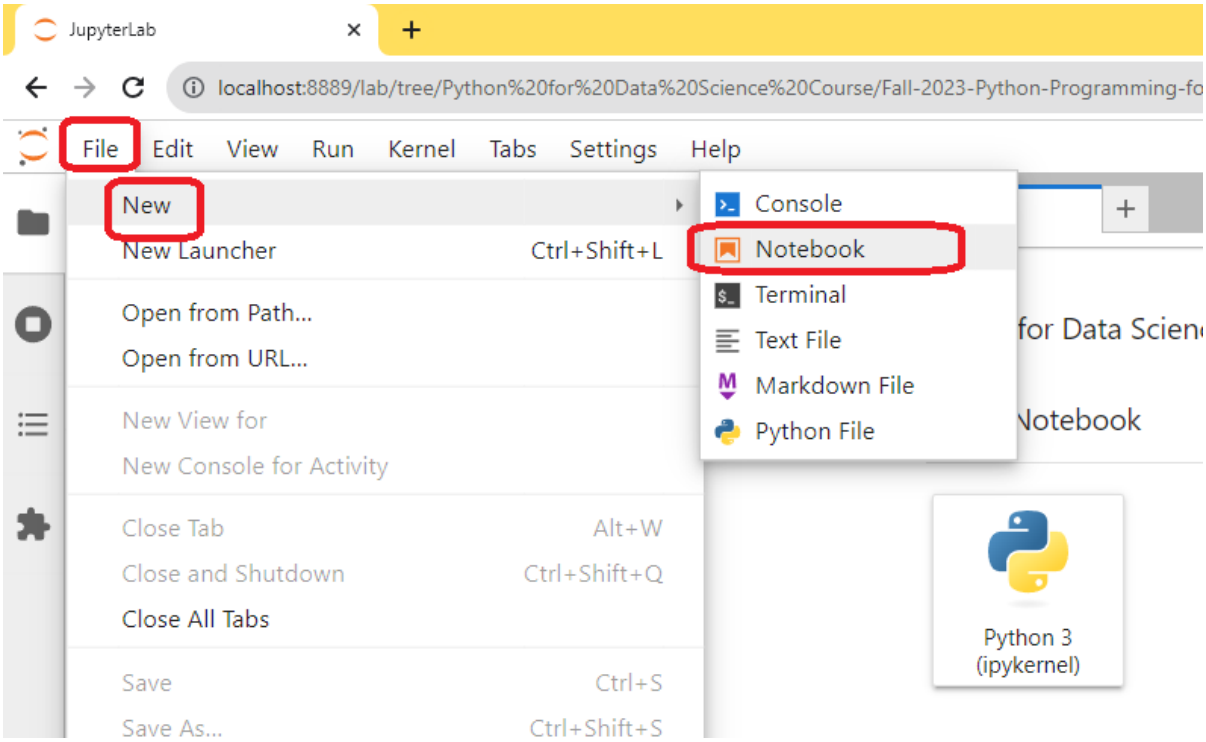


And probably the most convenient way is to create a shortcut to Jupyter Lab on your desktop for direct access.

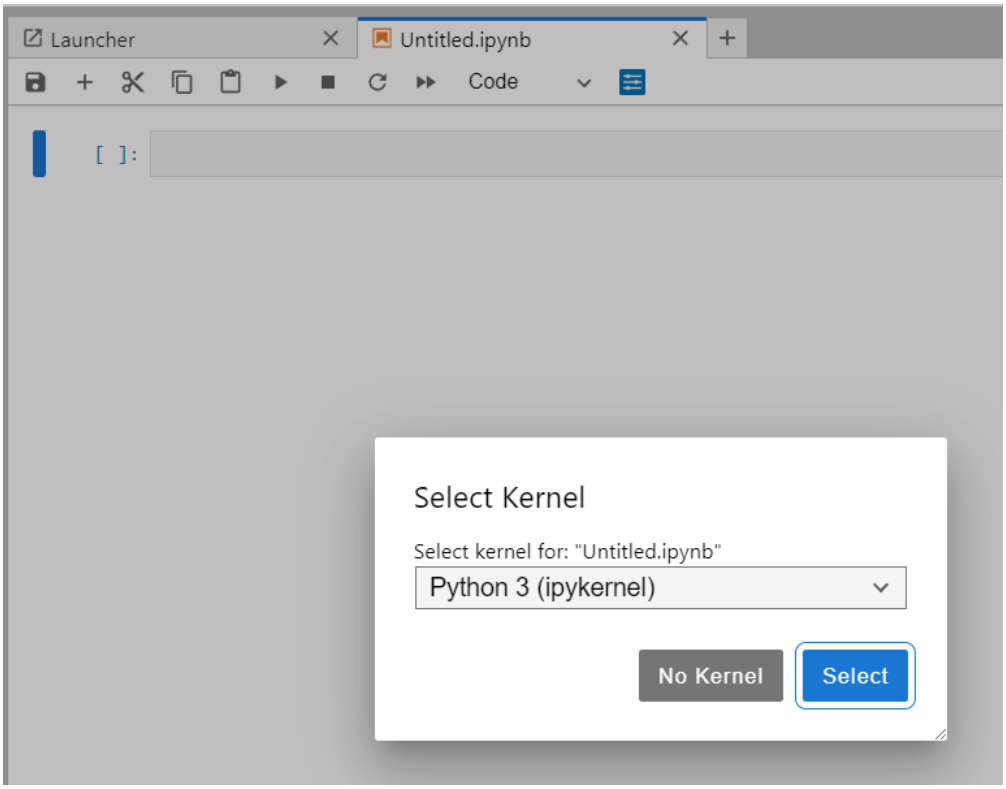
Creating a New Notebook

Now, let's create a new notebook.

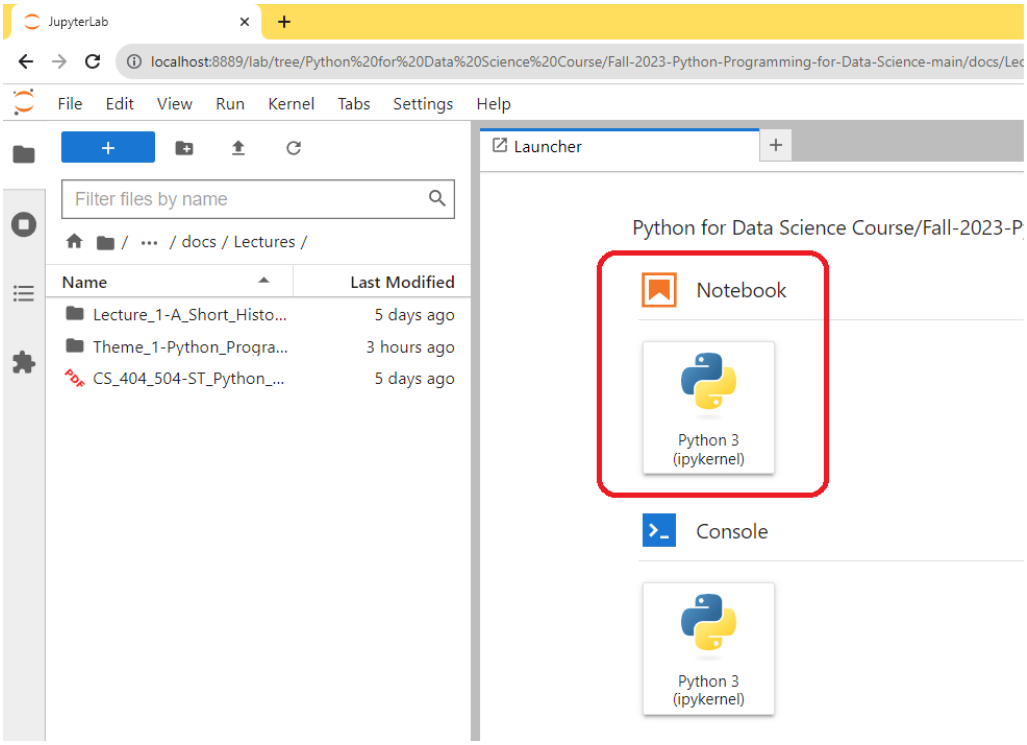
One way to do that is to click on the `File` tab in the top menu, then select the `New` button, and in the drop-down menu choose `Notebook`. Note that the `New` button also have options to open a new console or terminal (which allows to run shell commands directly in your browser, instead of the Windows Command Prompt), or create a new text file, markdown file, or Python file.



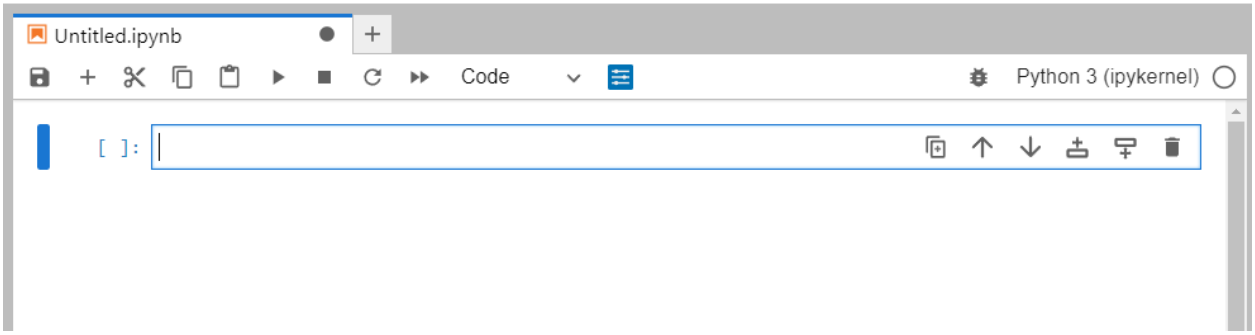
In the newly opened window you will be prompted to select the kernel for the notebook. We can select the default Python 3 kernel. The next section explains more about the kernels in notebooks.



Or, an even simpler way for creating a new notebook, is to directly click on `Notebook Python 3 (ipykernel)` icon on the Jupyter Dashboard.

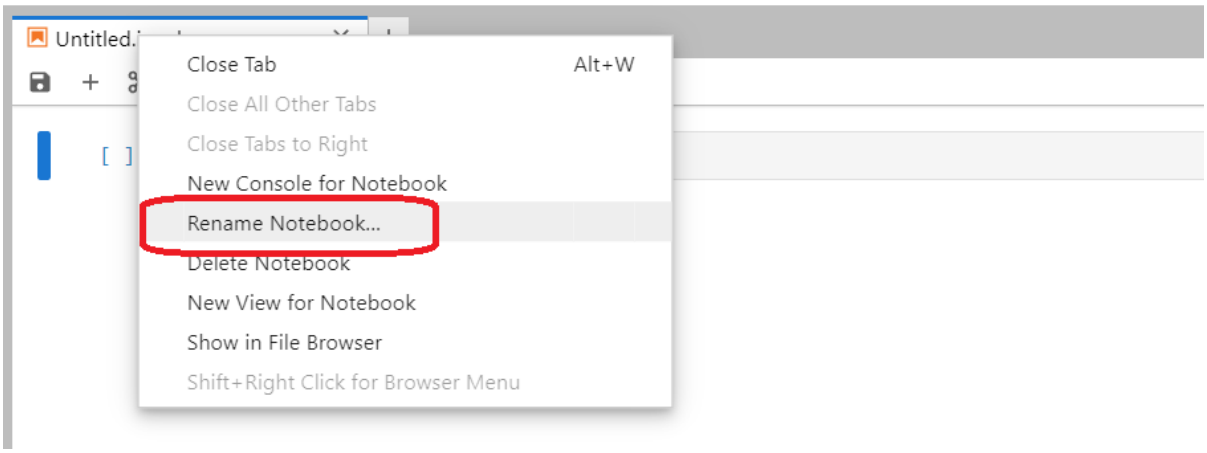


The newly created notebook will look like this.



The default assigned title to new notebooks is `Untitled`. You can see the title at the top of the page.

To change the title into a more descriptive one, right-click on the word `Untitled`, in the drop-down menu select `Rename Notebook..`, and enter the name `Hello_world`, for example.



The new title of the Notebook should now show at the top of the page. Also, in the left-side panel you will see the new notebook `Hello_world.ipynb` listed in the current working directory.

The extension for Jupyter Notebook documents is `.ipynb`, which is an acronym for IPython Notebook.

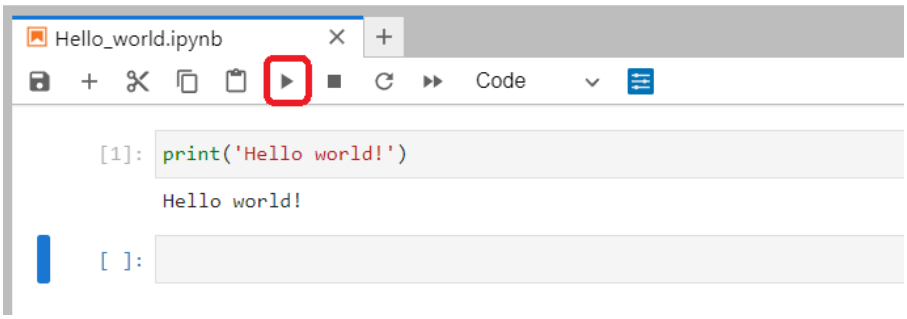
Cells in Jupyter Notebooks

Jupyter Notebooks organize codes in cells. Our newly created notebook has one empty cell in it.

The cells use the Python 3 kernel that we chose when we created the notebook. The used kernel for Python code execution in Jupyter Notebooks is also called IPython kernel. The kernel allows to execute Python code in the cell.

> A **cell** is a container for code that is to be executed by the notebook's kernel, or text that is to be displayed in the notebook.
>
> A **kernel** is a program that executes the code in a cell. Jupyter Notebook has a kernel for Python code, but also there are other kernels available for other programming languages.

Let's give it a try and write `print('Hello world!')` in the cell. To execute the cell, we can either click on the `Run button` in the toolbar on the top of the screen, or we can press the `Shift` and `Enter` keys on the keyboard.

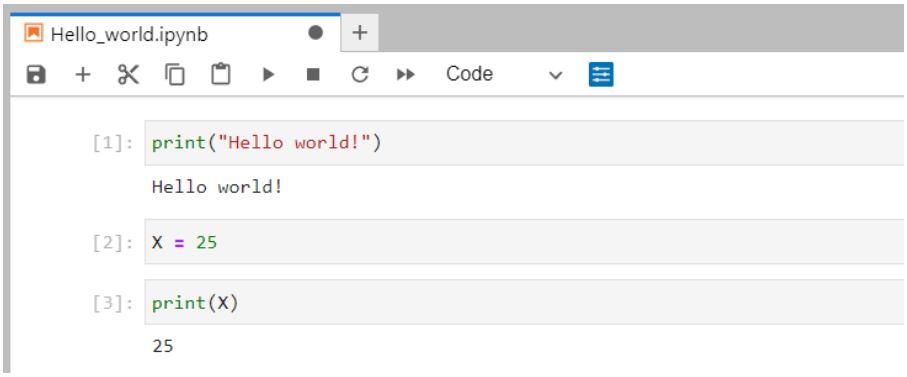


When you run a code cell, the kernel executes the code and the output of the code is returned back to the cell to be displayed.

Notice that each cell has a number, enclosed in square brackets `[1]` to the left of the cell. If we write another statement in the next cell, the order of the cell will be automatically changed to `[2]`. But also, if we run the first cell multiple times, each time the cell is executed the cell number will increase. This way, in a program with many cells, we can tell the order in which the cells were executed.

Using multiple cells in a module allows to separate the code into logical groups for improved code readability.

Furthermore, the variables and imported packages are shared across cells. For instance, if we define a variable `X` in cell 2, we can invoke the same variable in cell 3. Therefore, importing libraries or defining functions needs to be done only once in a notebook, and they are afterward shared by all cells in the notebook.



One difference between a Python script and a Jupyter notebook is that we can run the cells in a notebook out of order. Because of this, we should be very careful about variable reuse across cells. A good practice is to arrange the cells in the order in which we expect others to run the cells when they use the notebook.

If there are errors in our code in a Jupyter Notebook that can be related to variable name conflicts, it is recommended to select `Restart Kernel...` from the `Kernel` tab. This will restart the kernel and clear all variables.

Before sharing notebooks with others, or before submitting your homework assignment in a Jupyter notebook, it is a good practice to select `Restart Kernal and Run All Cells...` from the `Kernel` tab to avoid issues with variable reuse across cells.

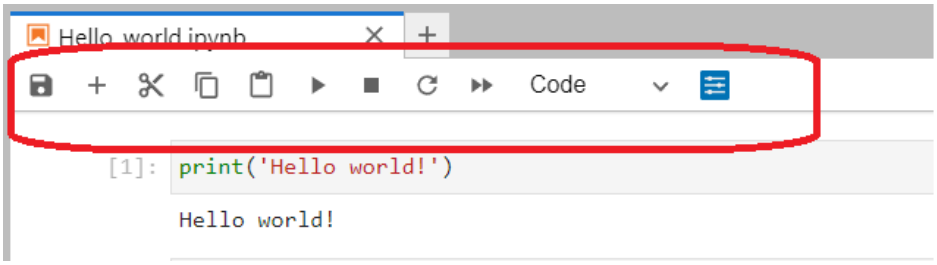
7.28.2 Jupyter Lab User Interface

Drop-Down Menus

Jupyter Notebooks have the following drop-down menus on the top of the screen:

- **File**: allows to create a new notebook, open existing notebooks, or save, download, and close notebooks.
- **Edit**: have cut, copy, and paste cells buttons, and also allows to delete, split, merge, or reorder the cells in a notebook.
- **View**: offers commands for toggling the visibility of the header, toolbar, and line numbers.
- **Run**: have various commands for running cells above or below a cell.
- **Kernel**: is for working with the selected kernel, allowing to restart the kernel, reconnect to the kernel, shut it down, or change the kernel.
- **Tabs**: provides a tabbed interface to manage and navigate open documents and interfaces.
- **Settings**: has access to configuration options and preferences for customizing Jupyter Lab.
- **Help**: to get help about Jupyter Notebooks, learn about keyboard shortcuts, or access links to reference materials.

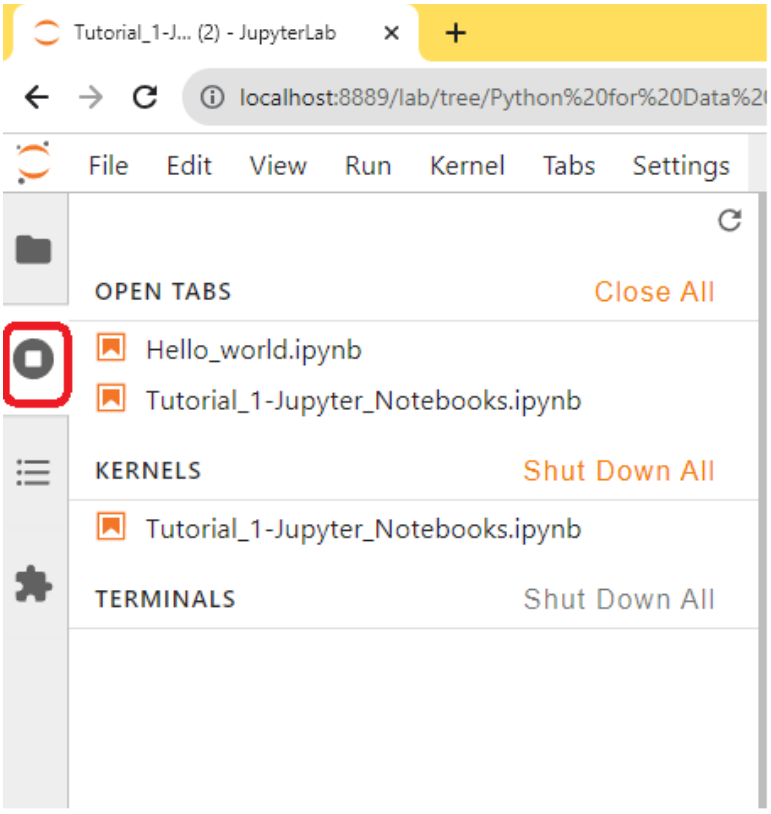
Beside the top menus, several functions that are the most commonly used can be invoked from the top toolbar. If you hover the mouse over the icons, a brief description will show up. The icons include: save, insert a new cell, cut, copy, and paste selected cells, run cells, interrupt the kernel, restart the kernel, select the cell type, and render with panel.



Left Sidebar in Jupyter Lab

The **left side-bar** or the left-side panel in Jupyter Lab contains several tabs, including:

- **File browser**: shows the content of the current working directory.
- **Running terminals and kernels**: lists the terminals and notebooks that are currently running (shown below). It allows to shut down notebooks and release computational resources.
- **Table of contents**: allows to navigate the structure of the notebook.
- **Extension manager**: for managing third party-extensions.



Shutting Down Notebooks

Note that even if we close the tab of a notebook, the kernel will continue to run in the background, and it needs to be shut down from the *Running terminals and kernels* tab shown in the above figure to be fully "closed".

Otherwise, Jupyter Notebooks are auto-saved pretty frequently (every 120 seconds), and it is rare to lose data .When the notebook is saved, A checkpoint file is created in a subdirectory of the working directory named `.ipynb_checkpoints`. The checkpoint file enables to recoveranyr unsaved work in the event of an unexpected issue.

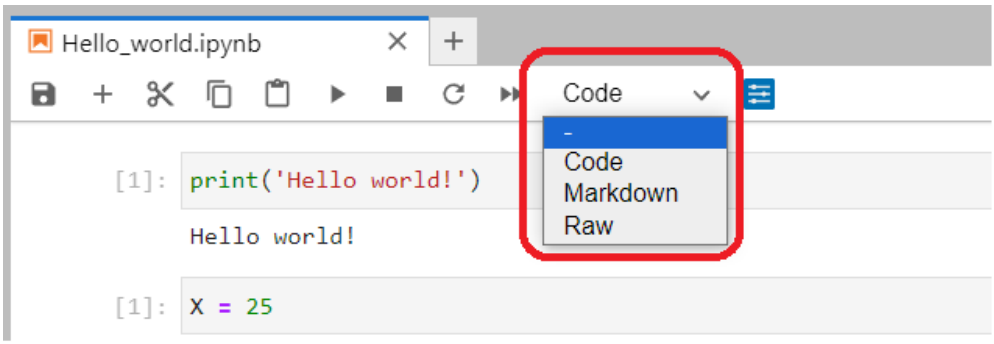
Cell Types

There are three cell types in Jupyter Notebooks: Code, Markdown, and Raw.

The default one is **Code**, which we use for running codes.

Markdown cells are used for Markdown, which is a markup language that is a superset of HTML. In fact, I created this file using Markdown language cells. If you click within the cells of this file, you can see the original markdown text. Markdown is a simple language, that adds formatting elements to plain text. For example, headings are created by adding the # mark (as in `# Lecture 1`), for **bold font** use two asterisks, for *italic font* use a single asterisk, etc. Still, Markdown accepts standard HTML language, which can add complexity when needed. See several examples in the next section, and to learn more about the syntax of Markdown language please visit follow this link.

Raw cell type is only intended for special use cases, and it allows using the nbconvert command line tool to control the formatting when converting a Notebook to another format.



Markdown Language

The following cell shows examples of Markdown language code, and the cell afterward displays the output of the cell. Note that unlike the code cells that have line numbers on the left, the markdown cells don't have line numbers.

```
### Example Heading: This is Level 3 Heading
This is some plain text that forms a paragraph.
Add emphasis via **bold** and __bold__, *italic* and _italic_, or ***bold and italic***␣
↪and ___bold and italic___.
Paragraphs must be separated by an empty line.
- Sometimes we want to include lists.
    - Which can be indented.

[It is possible to include hyperlinks](https://www.example.com).

1. Lists can also be numbered.
2. For ordered lists.
```

(continues on next page)

(continued from previous page)

```
We can add inline equations using LaTeX code, e.g., $c = \sqrt{a^2 + b^2}$,
and equations in a new line: $$c = \sqrt{a^2 + b^2}$$

Inline code uses single backticks: `foo()`, and code blocks use triple backticks or they␣
↪can be indented by 4 spaces:

    foo()

And finally, adding images is easy: ![Image title text](images/house.png)

# This is a level 1 heading example
## This is a level 2 heading example
```

Example Heading: This is Level 3 Heading

This is some plain text that forms a paragraph. Add emphasis via **bold** and **bold**, *italic* and *italic*, or **bold and italic** and **bold and italic**. Paragraphs must be separated by an empty line. - Sometimes we want to include lists. - Which can be indented.

It is possible to include hyperlinks.

1. Lists can also be numbered.
2. For ordered lists.

We can add inline equations using LaTeX code, e.g., $c = \sqrt{a^2 + b^2}$, and equations in a new line:

$$c = \sqrt{a^2 + b^2}$$

Inline code uses single backticks: `foo()`, and code blocks use triple backticks:

```
bar()
```

Or can be indented by 4 spaces:

```
foo()
```

And finally, adding images is easy:

Summary

In conclusion, Jupyter Notebooks are very useful for learning Python and testing your codes, as well as for sharing Python codes because others can directly see the outputs of the codes, which may include numerical results, graphs, tables, and other visualizations. The notebooks also display any error messages and other important information in the code. For instance, when training neural networks it can sometimes take hours for executing the code. Being able to see all the results and outputs from the models without the need to run the codes is extremely helpful and simplifies collaboration with others.

The following are tips for best practices when working with Jupyter Notebooks:

- Don't forget to name your notebooks, and don't have several notebooks `Untitled`, `Untitled (1)`, etc. in each folder.
- Provide comments to your code to improve the code readability, and help others understand your code.
- Arrange your code into cells, using logical grouping of the code lines.
- Keep the cells simple, and don't put too many functions into one cell.
- Try to import all packages in the first code cell of the notebook.
- Display the graphs and plots inline, so that they are visible to others.

BACK TO TOP

7.29 Tutorial 2 - Terminal and Command Line

7.29.1 Introduction to Command Line Interface

Command Line Interface (CLI) is a text-based interface that allows users to interact with a computer system by typing commands.

It's a way to give instructions to the computer using textual input. The command line is used to execute various commands, run programs, navigate the file system, and perform various tasks.

It's commonly found in operating systems like Unix, Linux, and Windows, where users can enter commands directly into the command-line interface.

Why use CLI?

- CLI is more efficient for certain tasks, since we can execute the task with a single command instead of multiple clicks on different windows.
- Although Windows has features like Windows Task Scheduler, CLI is more flexible and easier to automate tasks.
- For some tasks, such as running tasks on a remote server, CLI is the only way to interact with a computer.

Basic Definitions Related to CLI

Terminal: A terminal is a software application that offers access to a CLI. It's the platform or environment in which the command line is displayed and utilized. Essentially, it's the visual medium that facilitates communication with the computer via the command line.

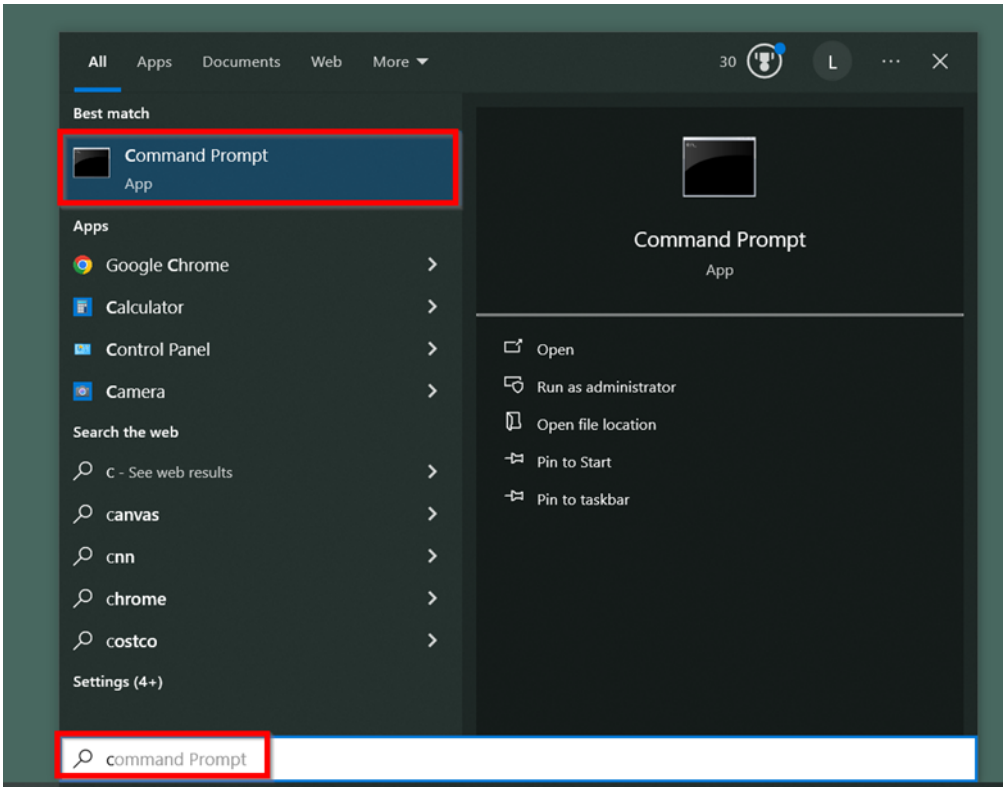
Shell: A shell is a program designed to interpret and execute the commands entered by users through the command line. Acting as a command-line interpreter, it translates human-readable commands into instructions comprehensible by the operating system for execution. The shell also manages tasks like input and output redirection, environment variables, scripting, and various other functionalities. Different operating systems deploy distinct shell programs. For example, Unix-like systems typically utilize shells such as Bash, Zsh, or Fish, while Windows employs the Command Prompt and PowerShell.

Console: The console serves as the physical device that enables you to interact with the computer.

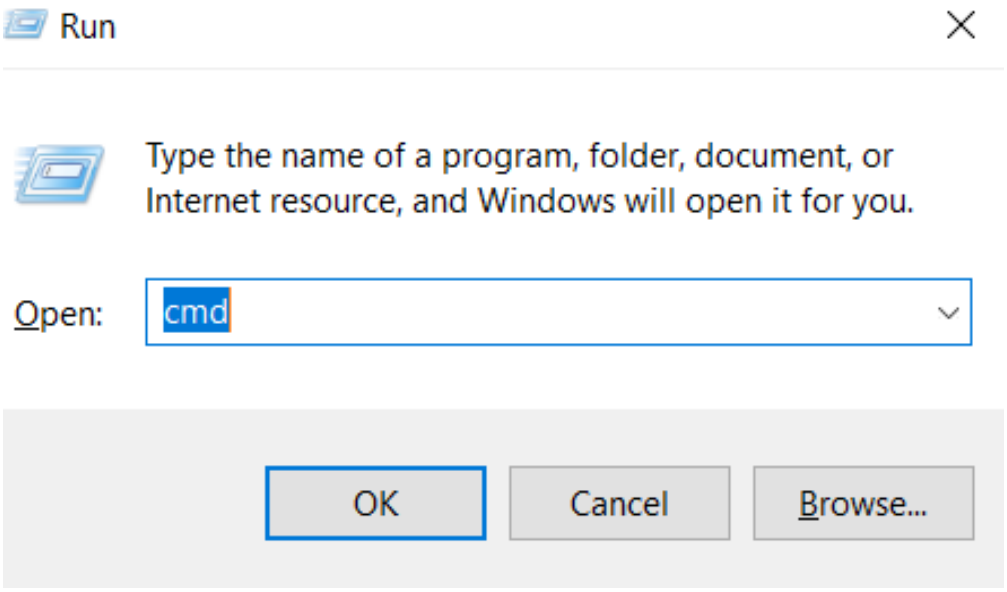
In summary, the command line functions as a text-based instruction method, the terminal provides the visual interface for the command line, the shell is the program responsible for interpreting and executing input commands through the command line, and the console is the computer screen, keyboard, mouse we use to interact with our computer.

Access the Command Prompt in Windows

- To access the Command Prompt, enter "command prompt" in the Start menu's search bar to search for it. Alternatively, you can enter "cmd", which is the abbreviated name of the executable responsible for running the Command Prompt.

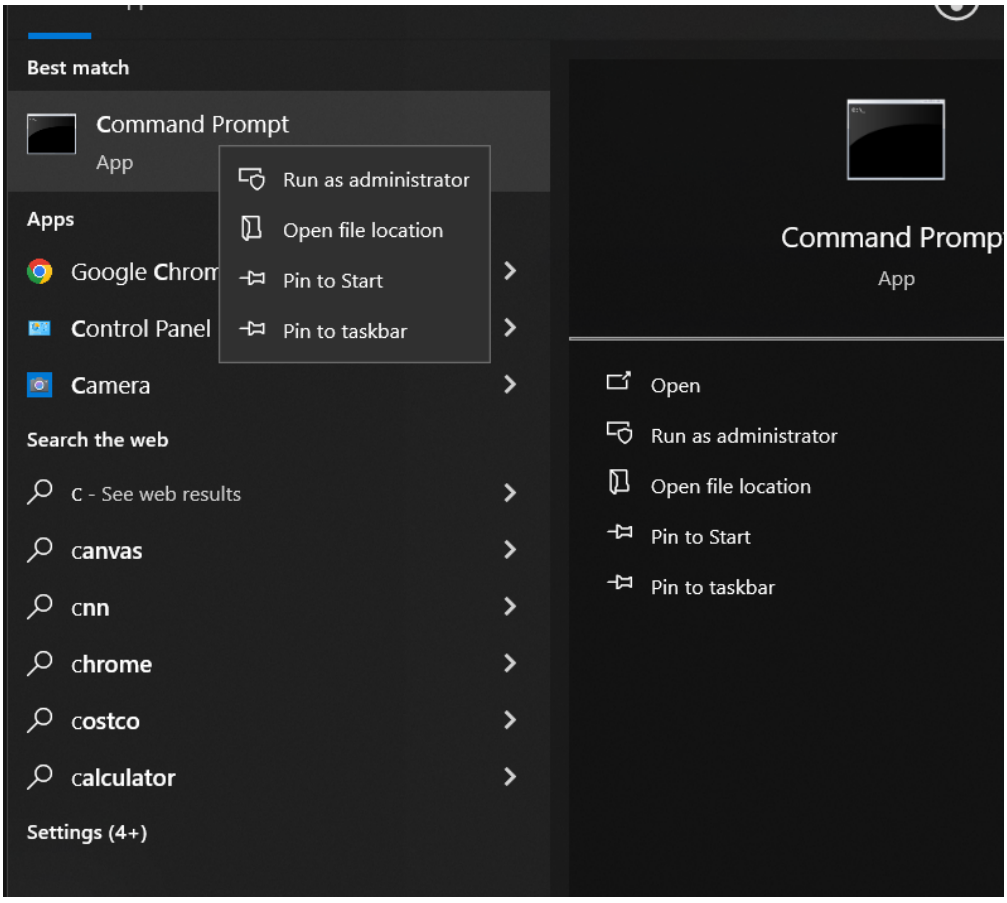


- Or, press Win + R to open the Run box, then enter "cmd" and press Enter.



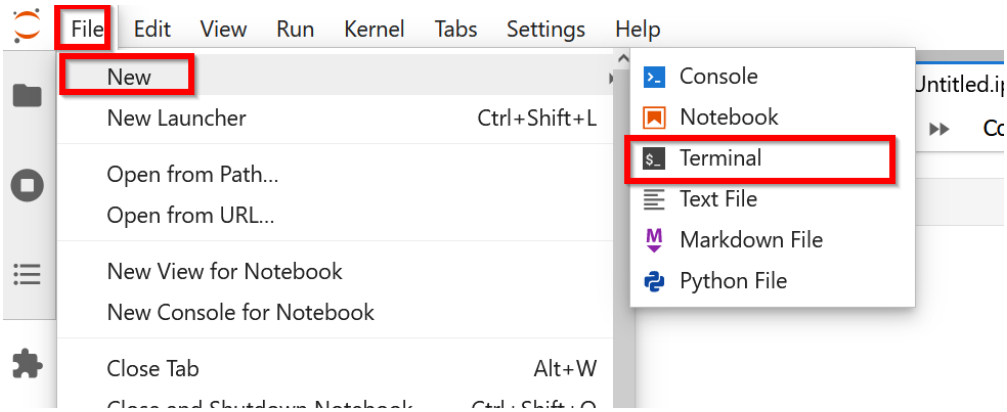
- Alternatively, press Win + X, and choose Command Prompt (or Powershell, depending on Windows settings) from the menu.

To run Command Prompt as an admin, hold Ctrl+Shift before pressing Enter, or right-click on the Command Prompt to select "Run as administrator".

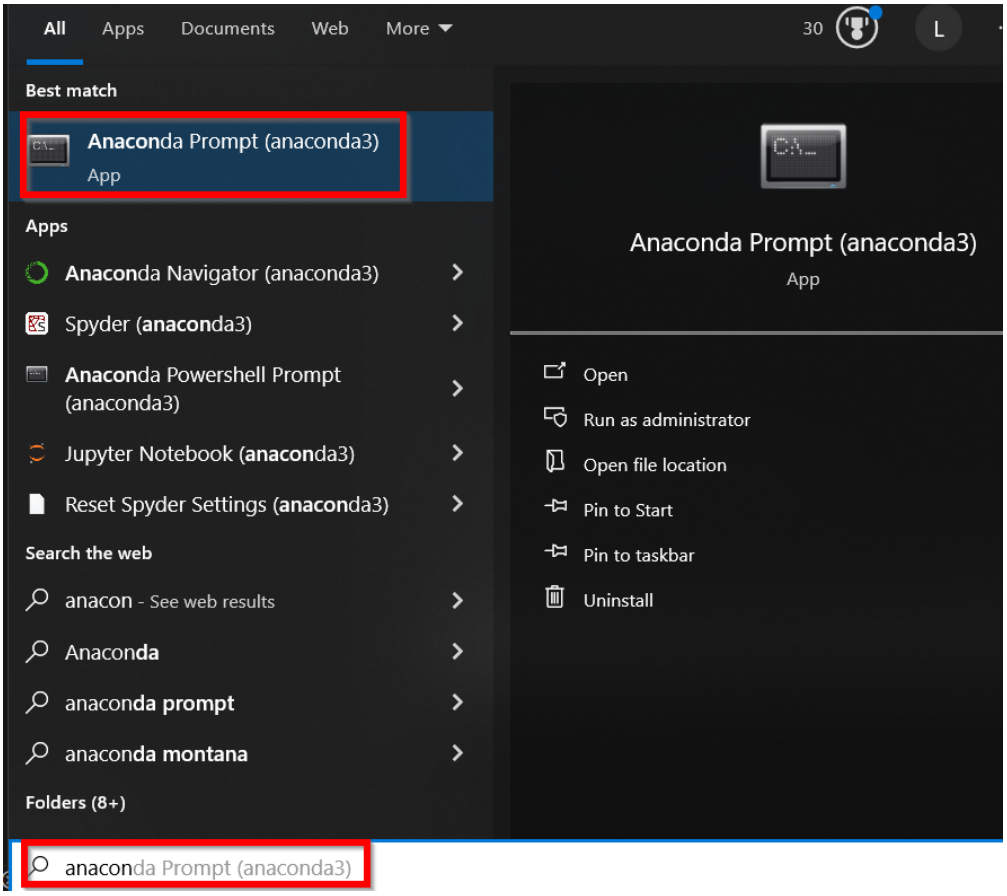


Other Ways to Access CLI in Windows

- Access through **Jupyter Lab**. After opening Jupyter Lab, in the top menu select the **File** tab, then click **New**, and select **Terminal**.



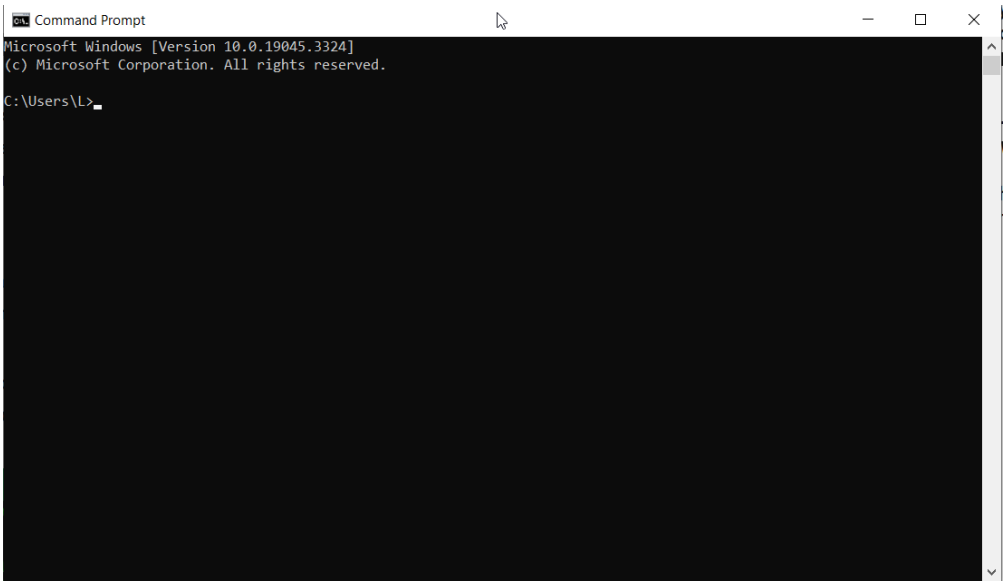
- Or, enter **Anaconda Prompt** in the Start menu's search bar to search for it. The difference between Anaconda Prompt and Command Prompt is that Anaconda Prompt offers some conda-specific features, such as conda environment management, package management, and pre-configured data science packages.



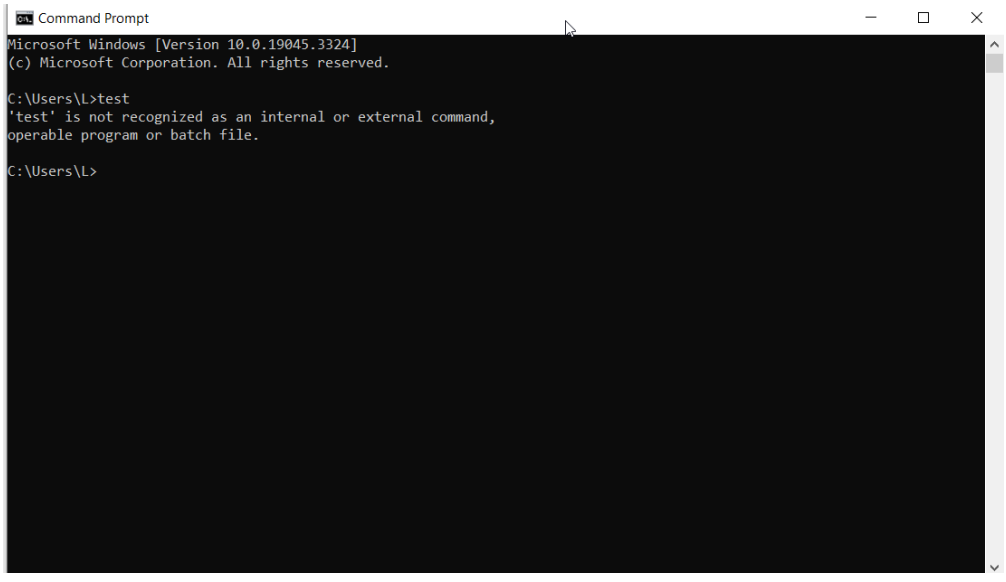
7.29.2 Command Prompt Basics

After opening a Command Prompt window, some basic information will be displayed.

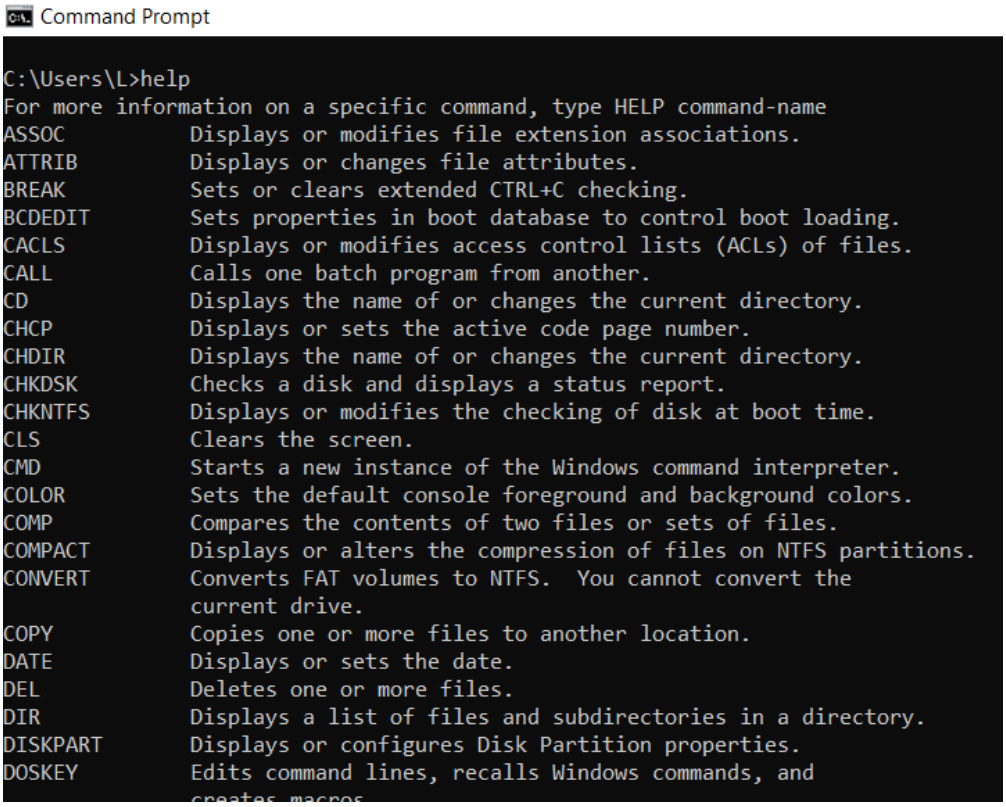
For example, the current directory `C:\Users\[Username]>` will be displayed as in the figure below.



Only valid commands will be accepted in the Command Prompt, otherwise a message like this will be displayed.



Help command: typing `help` in the Command Prompt will display a list of commands and the associated descriptions.



For more information on a specific command, type `HELP command-name`, e.g. `HELP DIR`. As you may have noticed, the commands in Windows Command Prompt are not case-sensitive, and both `help` and `HELP` work.

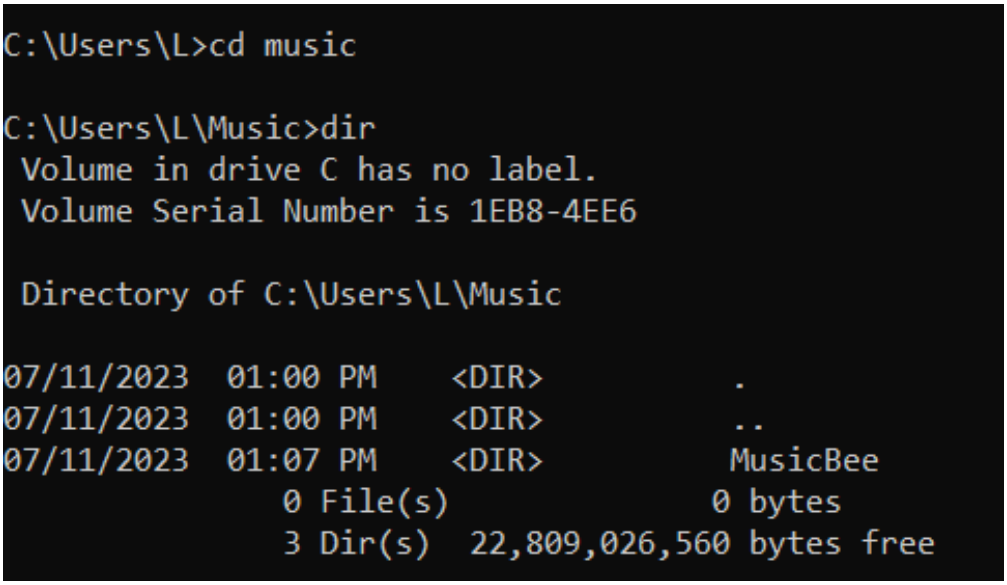
```
C:\Users\L>HELP DIR
Displays a list of files and subdirectories in a directory.

DIR [drive:][path][filename] [/A[[:]attributes]] [/B] [/C] [/D] [/L] [/N]
  [/O[[:]sortorder]] [/P] [/Q] [/R] [/S] [/T[[:]timefield]] [/W] [/X] [/4]

  [drive:][path][filename]
              Specifies drive, directory, and/or files to list.

  /A          Displays files with specified attributes.
  attributes   D  Directories                R  Read-only files
               H  Hidden files               A  Files ready for archiving
               S  System files               I  Not content indexed files
               L  Reparse Points             O  Offline files
               -  Prefix meaning not
  /B          Uses bare format (no heading information or summary).
  /C          Display the thousand separator in file sizes.  This is the
              default.  Use /-C to disable display of separator.
  /D          Same as wide but files are list sorted by column.
  /L          Uses lowercase.
  /N          New long list format where filenames are on the far right.
  /O          List by files in sorted order.
  sortorder    N  By name (alphabetic)       S  By size (smallest first)
               E  By extension (alphabetic)  D  By date/time (oldest first)
               G  Group directories first    -  Prefix to reverse order
  /P          Pauses after each screenful of information.
  /Q          Display the owner of the file.
  /R          Display alternate data streams of the file.
  /S          Displays files in specified directory and all subdirectories.
  /T          Controls which time field displayed or used for sorting
  timefield   C  Creation
              A  Last Access
              W  Last Written
  /W          Uses wide list format.
  /X          This displays the short names generated for non-8dot3 file
              names.  The format is that of /N with the short name inserted
              before the long name. If no short name is present, blanks are
              displayed in its place.
  /4          Displays four-digit years

Switches may be preset in the DIRCMD environment variable.  Override
preset switches by prefixing any switch with - (hyphen)--for example, /-W.
```

Listing and Changing Directories

`DIR` is an abbreviation for the word directory, and it lists the contents of the directory (folder) that you are currently in.

```
C:\Users\L>dir
 Volume in drive C has no label.
 Volume Serial Number is 1EB8-4EE6

 Directory of C:\Users\L

08/24/2023  06:43 PM    <DIR>          .
08/24/2023  06:43 PM    <DIR>          ..
05/14/2023  12:13 PM    <DIR>          .art
04/04/2023  11:29 AM    <DIR>          .cache
08/25/2023  04:46 AM    <DIR>          .conda
05/12/2023  10:30 AM                42 .condarc
02/10/2023  07:38 PM    <DIR>          .continuum
08/24/2023  06:34 PM    <DIR>          .ipynb_checkpoints
03/12/2023  08:45 PM    <DIR>          .ipython
08/24/2023  04:53 PM    <DIR>          .jupyter
02/10/2023  07:48 PM    <DIR>          .keras
02/10/2023  07:48 PM    <DIR>          .matplotlib
02/14/2023  05:36 PM    <DIR>          .ms-ad
06/26/2023  03:19 PM    <DIR>          .spyder-py3
06/15/2023  06:30 AM    <DIR>          .vscode
09/08/2022  04:30 PM    <DIR>          3D Objects
05/12/2023  10:31 AM    <DIR>          anaconda3
09/08/2022  04:30 PM    <DIR>          Contacts
08/29/2023  06:07 PM    <DIR>          Desktop
08/29/2023  07:49 AM    <DIR>          Documents
08/29/2023  06:53 PM    <DIR>          Downloads
03/13/2023  06:09 PM    <DIR>          dwhelper
09/08/2022  04:30 PM    <DIR>          Favorites
08/24/2023  06:43 PM             2,225 hello_world.ipynb
08/24/2023  05:30 PM           112,737 house.png
03/12/2023  08:46 PM    <DIR>          Jedi
09/08/2022  04:30 PM    <DIR>          Links
07/11/2023  01:00 PM    <DIR>          Music
09/08/2022  05:16 PM    <DIR>          OneDrive
09/08/2022  04:31 PM    <DIR>          Pictures
06/26/2023  03:30 PM    <DIR>          Postman
09/08/2022  04:30 PM    <DIR>          Saved Games
03/10/2023  08:03 AM    <DIR>          Searches
08/24/2023  06:36 PM               617 Untitled.ipynb
07/01/2023  07:23 PM    <DIR>          Videos
               4 File(s)        115,621 bytes
              31 Dir(s)  22,836,273,152 bytes free
```

To change directory, use `CD` followed by the name of the folder you want to access.



To go back to the parent directory, use `CD ..`

```
C:\Users\L\Music>cd ..

C:\Users\L>dir
 Volume in drive C has no label.
 Volume Serial Number is 1EB8-4EE6

 Directory of C:\Users\L

08/24/2023  06:43 PM    <DIR>          .
08/24/2023  06:43 PM    <DIR>          ..
05/14/2023  12:13 PM    <DIR>          .art
04/04/2023  11:29 AM    <DIR>          .cache
08/25/2023  04:46 AM    <DIR>          .conda
05/12/2023  10:30 AM                42 .condarc
02/10/2023  07:38 PM    <DIR>          .continuum
08/24/2023  06:34 PM    <DIR>          .ipynb_checkpoints
03/12/2023  08:45 PM    <DIR>          .ipython
08/24/2023  04:53 PM    <DIR>          .jupyter
02/10/2023  07:48 PM    <DIR>          .keras
02/10/2023  07:48 PM    <DIR>          .matplotlib
02/14/2023  05:36 PM    <DIR>          .ms-ad
06/26/2023  03:19 PM    <DIR>          .spyder-py3
06/15/2023  06:30 AM    <DIR>          .vscode
09/08/2022  04:30 PM    <DIR>          3D Objects
05/12/2023  10:31 AM    <DIR>          anaconda3
09/08/2022  04:30 PM    <DIR>          Contacts
08/29/2023  06:07 PM    <DIR>          Desktop
08/29/2023  07:49 AM    <DIR>          Documents
08/29/2023  06:53 PM    <DIR>          Downloads
03/13/2023  06:09 PM    <DIR>          dwhelper
09/08/2022  04:30 PM    <DIR>          Favorites
08/24/2023  06:43 PM             2,225 hello_world.ipynb
08/24/2023  05:30 PM           112,737 house.png
03/12/2023  08:46 PM    <DIR>          Jedi
09/08/2022  04:30 PM    <DIR>          Links
07/11/2023  01:00 PM    <DIR>          Music
09/08/2022  05:16 PM    <DIR>          OneDrive
09/08/2022  04:31 PM    <DIR>          Pictures
06/26/2023  03:30 PM    <DIR>          Postman
09/08/2022  04:30 PM    <DIR>          Saved Games
03/10/2023  08:03 AM    <DIR>          Searches
08/24/2023  06:36 PM               617 Untitled.ipynb
07/01/2023  07:23 PM    <DIR>          Videos
               4 File(s)        115,621 bytes
              31 Dir(s)  22,809,092,096 bytes free
```

Creating and Deleting Files/Folders in Command Prompt

Use `mkdir` followed by `[new folder name/new file name]` to create a new folder/file.

E.g., the command to create a folder named "test" is `mkdir test`.

```
C:\Users\L>mkdir test

C:\Users\L>dir
 Volume in drive C has no label.
 Volume Serial Number is 1EB8-4EE6

 Directory of C:\Users\L

08/30/2023  12:23 AM    <DIR>          .
08/30/2023  12:23 AM    <DIR>          ..
05/14/2023  12:13 PM    <DIR>          .art
04/04/2023  11:29 AM    <DIR>          .cache
08/25/2023  04:46 AM    <DIR>          .conda
05/12/2023  10:30 AM                42 .condarc
02/10/2023  07:38 PM    <DIR>          .continuum
08/24/2023  06:34 PM    <DIR>          .ipynb_checkpoints
03/12/2023  08:45 PM    <DIR>          .ipython
08/24/2023  04:53 PM    <DIR>          .jupyter
02/10/2023  07:48 PM    <DIR>          .keras
02/10/2023  07:48 PM    <DIR>          .matplotlib
02/14/2023  05:36 PM    <DIR>          .ms-ad
06/26/2023  03:19 PM    <DIR>          .spyder-py3
06/15/2023  06:30 AM    <DIR>          .vscode
09/08/2022  04:30 PM    <DIR>          3D Objects
05/12/2023  10:31 AM    <DIR>          anaconda3
09/08/2022  04:30 PM    <DIR>          Contacts
08/29/2023  06:07 PM    <DIR>          Desktop
08/29/2023  07:49 AM    <DIR>          Documents
08/29/2023  06:53 PM    <DIR>          Downloads
03/13/2023  06:09 PM    <DIR>          dwhelper
09/08/2022  04:30 PM    <DIR>          Favorites
08/24/2023  06:43 PM             2,225 hello_world.ipynb
08/24/2023  05:30 PM           112,737 house.png
03/12/2023  08:46 PM    <DIR>          Jedi
09/08/2022  04:30 PM    <DIR>          Links
07/11/2023  01:00 PM    <DIR>          Music
09/08/2022  05:16 PM    <DIR>          OneDrive
09/08/2022  04:31 PM    <DIR>          Pictures
06/26/2023  03:30 PM    <DIR>          Postman
09/08/2022  04:30 PM    <DIR>          Saved Games
03/10/2023  08:03 AM    <DIR>          Searches
08/30/2023  12:23 AM    <DIR>          test
08/24/2023  06:36 PM               617 Untitled.ipynb
07/01/2023  07:23 PM    <DIR>          Videos
               4 File(s)        115,621 bytes
              32 Dir(s)  22,782,353,408 bytes free
```

Use `rmdir` followed by `[new folder name]` to delete a folder.

```
C:\Users\L>rmdir test

C:\Users\L>dir
 Volume in drive C has no label.
 Volume Serial Number is 1EB8-4EE6

 Directory of C:\Users\L

08/30/2023  12:30 AM    <DIR>          .
08/30/2023  12:30 AM    <DIR>          ..
05/14/2023  12:13 PM    <DIR>          .art
04/04/2023  11:29 AM    <DIR>          .cache
08/25/2023  04:46 AM    <DIR>          .conda
05/12/2023  10:30 AM                42 .condarc
02/10/2023  07:38 PM    <DIR>          .continuum
08/24/2023  06:34 PM    <DIR>          .ipynb_checkpoints
03/12/2023  08:45 PM    <DIR>          .ipython
08/24/2023  04:53 PM    <DIR>          .jupyter
02/10/2023  07:48 PM    <DIR>          .keras
02/10/2023  07:48 PM    <DIR>          .matplotlib
02/14/2023  05:36 PM    <DIR>          .ms-ad
06/26/2023  03:19 PM    <DIR>          .spyder-py3
06/15/2023  06:30 AM    <DIR>          .vscode
09/08/2022  04:30 PM    <DIR>          3D Objects
05/12/2023  10:31 AM    <DIR>          anaconda3
09/08/2022  04:30 PM    <DIR>          Contacts
08/29/2023  06:07 PM    <DIR>          Desktop
08/29/2023  07:49 AM    <DIR>          Documents
08/29/2023  06:53 PM    <DIR>          Downloads
03/13/2023  06:09 PM    <DIR>          dwhelper
09/08/2022  04:30 PM    <DIR>          Favorites
08/24/2023  06:43 PM             2,225 hello_world.ipynb
08/24/2023  05:30 PM           112,737 house.png
03/12/2023  08:46 PM    <DIR>          Jedi
09/08/2022  04:30 PM    <DIR>          Links
07/11/2023  01:00 PM    <DIR>          Music
09/08/2022  05:16 PM    <DIR>          OneDrive
09/08/2022  04:31 PM    <DIR>          Pictures
06/26/2023  03:30 PM    <DIR>          Postman
09/08/2022  04:30 PM    <DIR>          Saved Games
03/10/2023  08:03 AM    <DIR>          Searches
08/24/2023  06:36 PM               617 Untitled.ipynb
07/01/2023  07:23 PM    <DIR>          Videos
               4 File(s)        115,621 bytes
              31 Dir(s)  22,783,762,432 bytes free
```

Use `del` followed by `[new file name]` to delete a file.

```
C:\Users\L>mkdir file_test.txt

C:\Users\L>dir
 Volume in drive C has no label.
 Volume Serial Number is 1EB8-4EE6

 Directory of C:\Users\L

08/30/2023  12:32 AM    <DIR>          .
08/30/2023  12:32 AM    <DIR>          ..
05/14/2023  12:13 PM    <DIR>          .art
04/04/2023  11:29 AM    <DIR>          .cache
08/25/2023  04:46 AM    <DIR>          .conda
05/12/2023  10:30 AM                42 .condarc
02/10/2023  07:38 PM    <DIR>          .continuum
08/24/2023  06:34 PM    <DIR>          .ipynb_checkpoints
03/12/2023  08:45 PM    <DIR>          .ipython
08/24/2023  04:53 PM    <DIR>          .jupyter
02/10/2023  07:48 PM    <DIR>          .keras
02/10/2023  07:48 PM    <DIR>          .matplotlib
02/14/2023  05:36 PM    <DIR>          .ms-ad
06/26/2023  03:19 PM    <DIR>          .spyder-py3
06/15/2023  06:30 AM    <DIR>          .vscode
09/08/2022  04:30 PM    <DIR>          3D Objects
05/12/2023  10:31 AM    <DIR>          anaconda3
09/08/2022  04:30 PM    <DIR>          Contacts
08/29/2023  06:07 PM    <DIR>          Desktop
08/29/2023  07:49 AM    <DIR>          Documents
08/29/2023  06:53 PM    <DIR>          Downloads
03/13/2023  06:09 PM    <DIR>          dwhelper
09/08/2022  04:30 PM    <DIR>          Favorites
08/30/2023  12:32 AM    <DIR>          file_test.txt
08/24/2023  06:43 PM             2,225 hello_world.ipynb
08/24/2023  05:30 PM           112,737 house.png
03/12/2023  08:46 PM    <DIR>          Jedi
09/08/2022  04:30 PM    <DIR>          Links
07/11/2023  01:00 PM    <DIR>          Music
09/08/2022  05:16 PM    <DIR>          OneDrive
09/08/2022  04:31 PM    <DIR>          Pictures
06/26/2023  03:30 PM    <DIR>          Postman
09/08/2022  04:30 PM    <DIR>          Saved Games
03/10/2023  08:03 AM    <DIR>          Searches
08/24/2023  06:36 PM               617 Untitled.ipynb
07/01/2023  07:23 PM    <DIR>          Videos
               4 File(s)        115,621 bytes
              32 Dir(s)  22,785,261,568 bytes free

C:\Users\L>del file_test.txt
C:\Users\L\file_test.txt\*, Are you sure (Y/N)? y
```

CLI Management

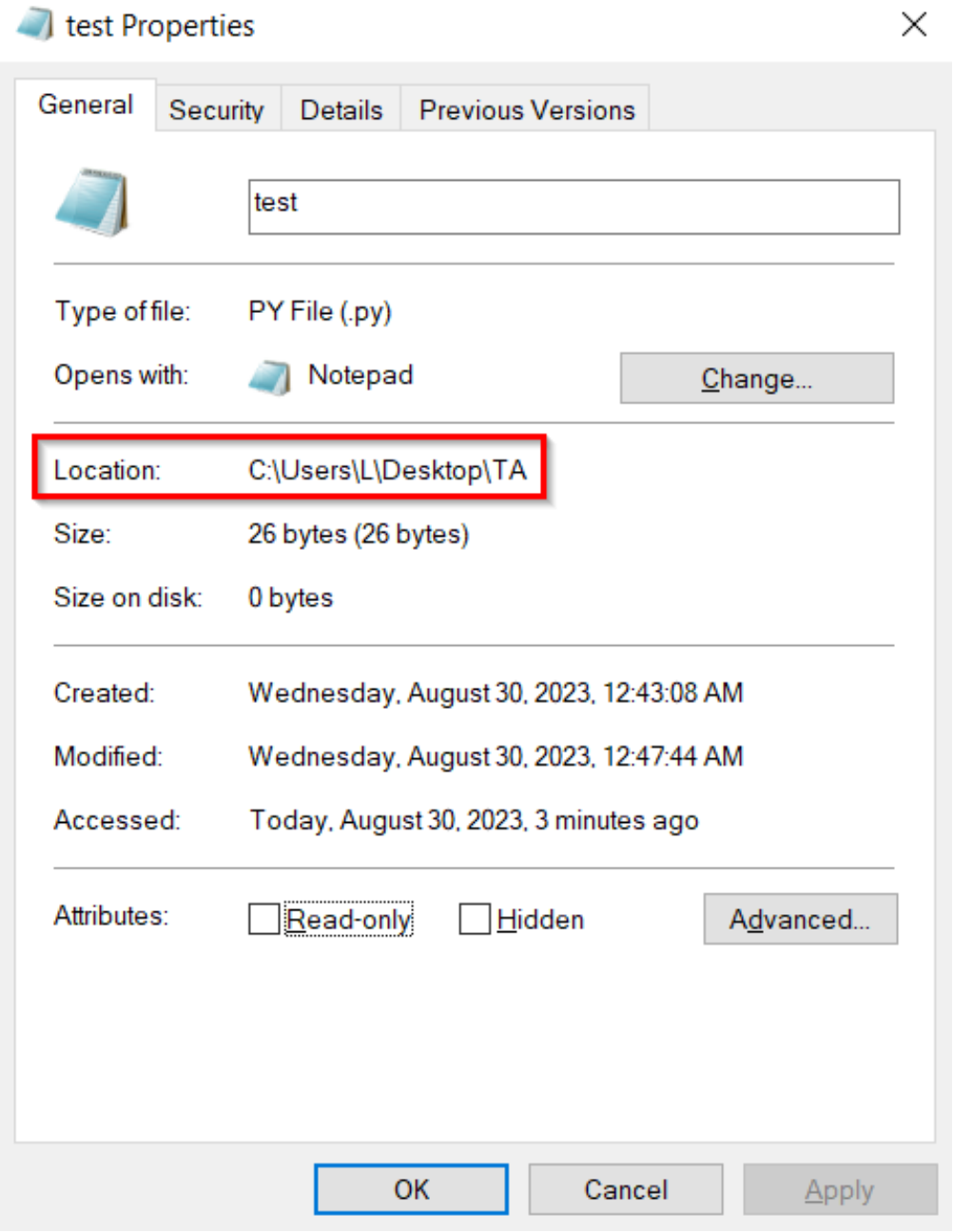
Enter `cls` to clear the previous content.

Press **Ctrl+C** to end a command that is running

Run Python Files in Command Prompt

To execute a Python file, simply type **python** followed by the path to the file.

To find the file directory, just right-click on the file, then click "Properties" and copy the directory of the folder.



The python file directory will be the name of the folder the file is in, followed by a backward slash \ and the file name.

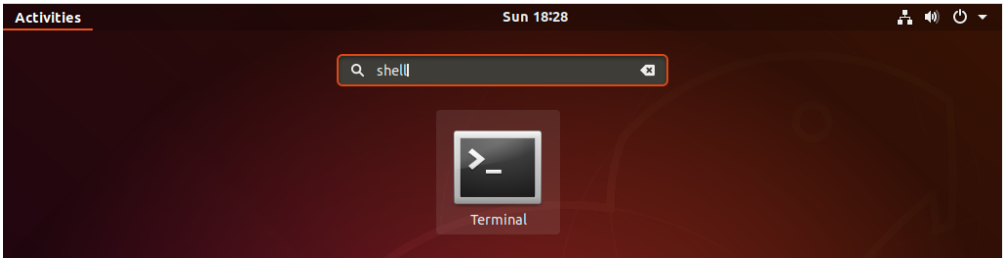
E.g., the file directory of the `test.py` file is `C:\Users\L\Desktop\TA`, and therefore the path for the python file is `C:\Users\L\Desktop\TA\test.py`.

To run this file, use the command `python "C:\Users\L\Desktop\TA\test.py"`.

```
C:\Users\L\Desktop\TA>python "C:\Users\L\Desktop\TA\test.py"
test command line

C:\Users\L\Desktop\TA>
```

Access the Terminal in Ubuntu (Linux)

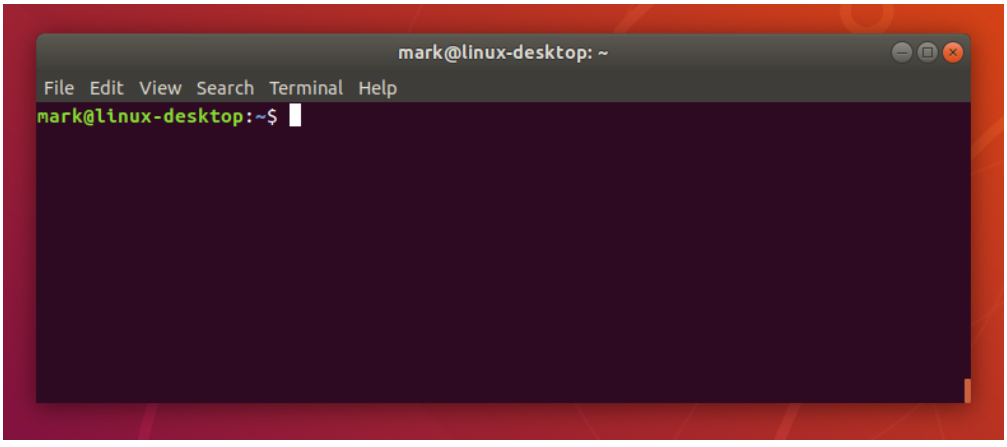
- Click on the Activities on the upper left of the screen, and type keywords like **"terminal"**, **"command"**, **"prompt"**, or **"shell"**.



- Press **Ctrl+Alt+T**.

Terminal Basics

After opening a terminal window, **username@linux-desktop:~$** will be displayed.



Basic Commands in Ubuntu (Linux)

Commands in Linux are case-sensitive, but commands in Windows are NOT case-sensitive.

The command `help` in Linux is the same command as in Windows, and it displays the available command names.

Similarly, `help` followed by a command-name will show the specific information of that command.

In Linux, `ls` is used to list the files in a directory instead of the command `dir` in Windows.

To run a python file in Linux terminal, type `python3` followed by the file name.

References

1. "A Beginner's Guide to the Windows Command Prompt" by Ben Stegner, available at: https://www.makeuseof.com/tag/a-beginners-guide-to-the-windows-command-line/.
2. "Command Line for Beginners – How to Use the Terminal Like a Pro" by German Cocca, available at: https://www.freecodecamp.org/news/command-line-for-beginners/.

BACK TO TOP

7.30 Tutorial 3 - Python IDEs, VS Code

This tutorial first presents several common environments and editors for Python code development, and afterward describes VS Code in more detail.

7.30.1 Python IDEs

> An **IDE (or Integrated Development Environment)** is a program that integrates tools to facilitate software development, such as a code editor, tools for program execution and debugging, and source control.

In other words, an IDE is a graphical user interface for program development, that allows end-users to edit, run, browse, and debug programs from a single interface. Using an IDE is especially convenient for beginners to start editing and running codes.

Most IDEs support several different programming languages. There are some that are designed specifically for Python development (like the IDLE described below).

Also, most IDEs offer a **code editor**, although the code editor can be an independent software application. A code editor can be a simple text editor, or it can also allow code execution and debugging. In comparison to IDEs, code editors are simpler, but they are also less resource-demanding.

In general, IDEs offer the following basic features:

- Save and reload code files
- Run code from within the environment
- Debugging support
- Syntax highlighting
- Automatic code formatting

The debugging support typically allows setting breakpoints in the code that stop the execution, showing variable values, and so on. For simpler debugging operations, most IDEs allow clicking on the text of an error message to quickly jump to the line of code where the error occurred, and after correcting the error, to quickly run the code.

Syntax highlighting refers to displaying source code using different colors and fonts according to the category of the written text and commands. It facilitates code writing by providing visually distinct syntax, and it can help to identify mistakes in the code.

Automatic code formatting can include features such as automated indentation, automated spacing around operators, auto-completion, etc.

Other helpful features often include: advanced text and file search options, help links or pop-up windows when hovering with the mouse over an object, selection list for object attributes (e.g., after a `.` press the `Tab` key), links to definition of objects, and similar.

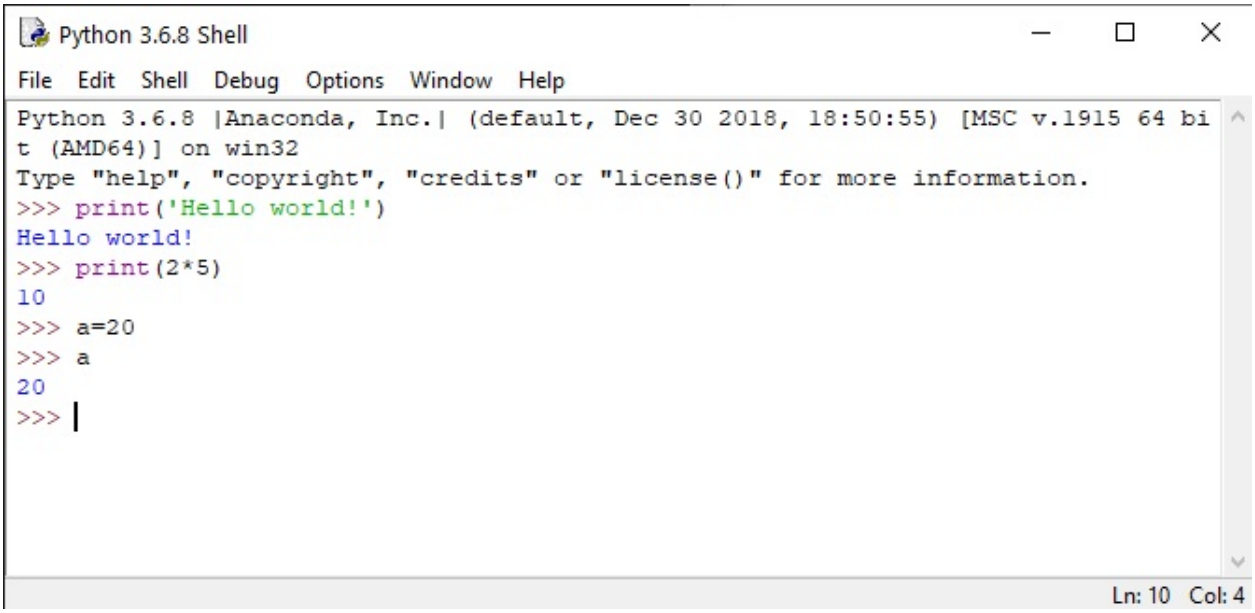
IDLE

IDLE is the original IDE for Python development. It is available for Windows, Unix, and Mac OS platforms, and it comes preinstalled with the Python installation. IDLE is free, easy to use, and portable across platforms.

The name is a variation of IDE, and it is named after one of the members of the Monty Python group - Eric Idle.

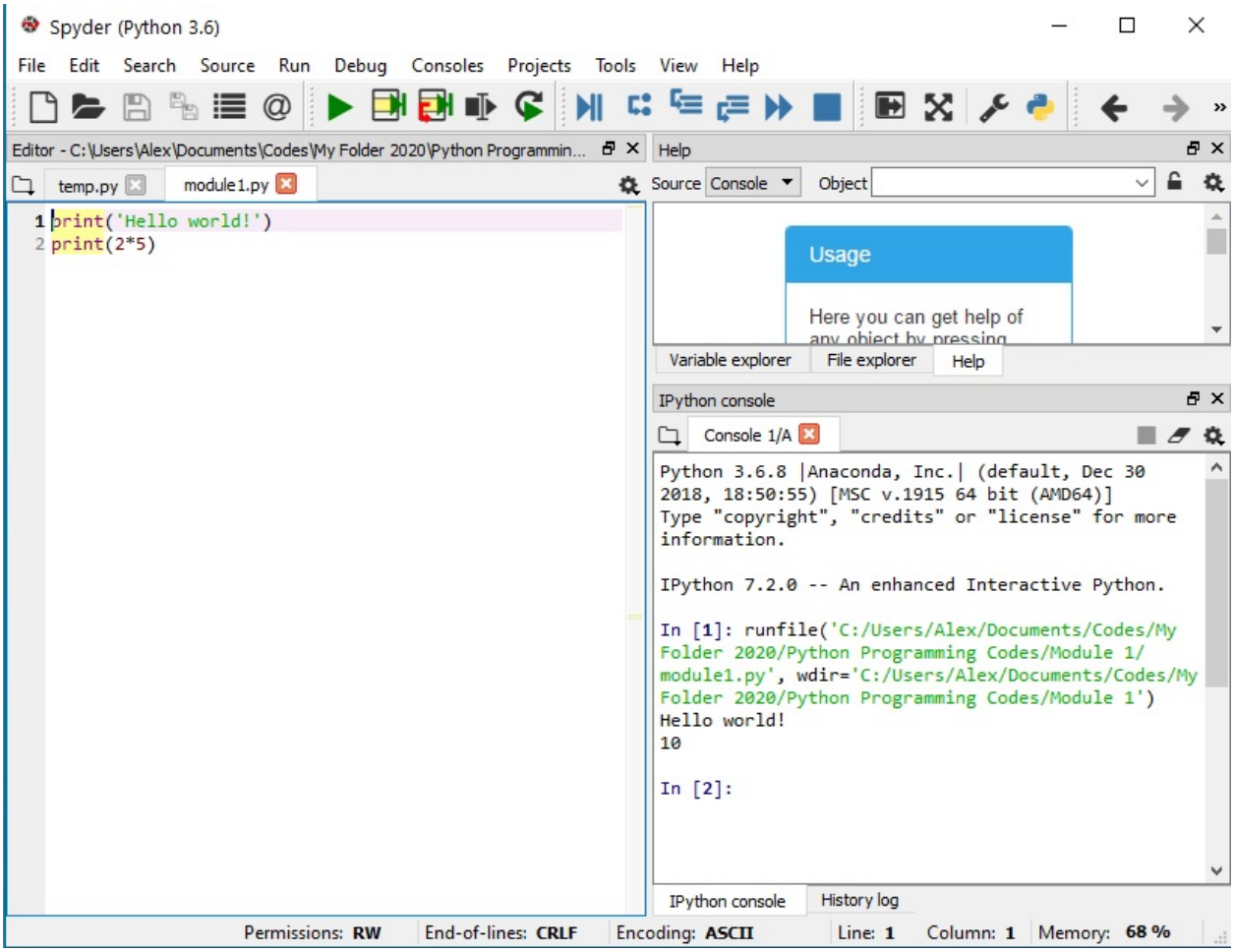
To access IDLE, type in the Windows Command Prompt: `python -m idlelib.idle`. The IDLE window is shown in the figure below. Notice the >>> prompt in IDLE, which allows interactive programming, i.e., we can type commands and run them immediately. Also note the syntax highlighting of the entered Python statements, and the menu bar on the top of the window that is similar to the Jupyter Notebook menu and to other MS Office apps (e.g., MS Word).

IDLE is simple to use and recommended for newcomers because it simplifies some details and does not assume prior experience with system command lines. On the other hand, it is somewhat limited compared to more advanced commercial IDEs.



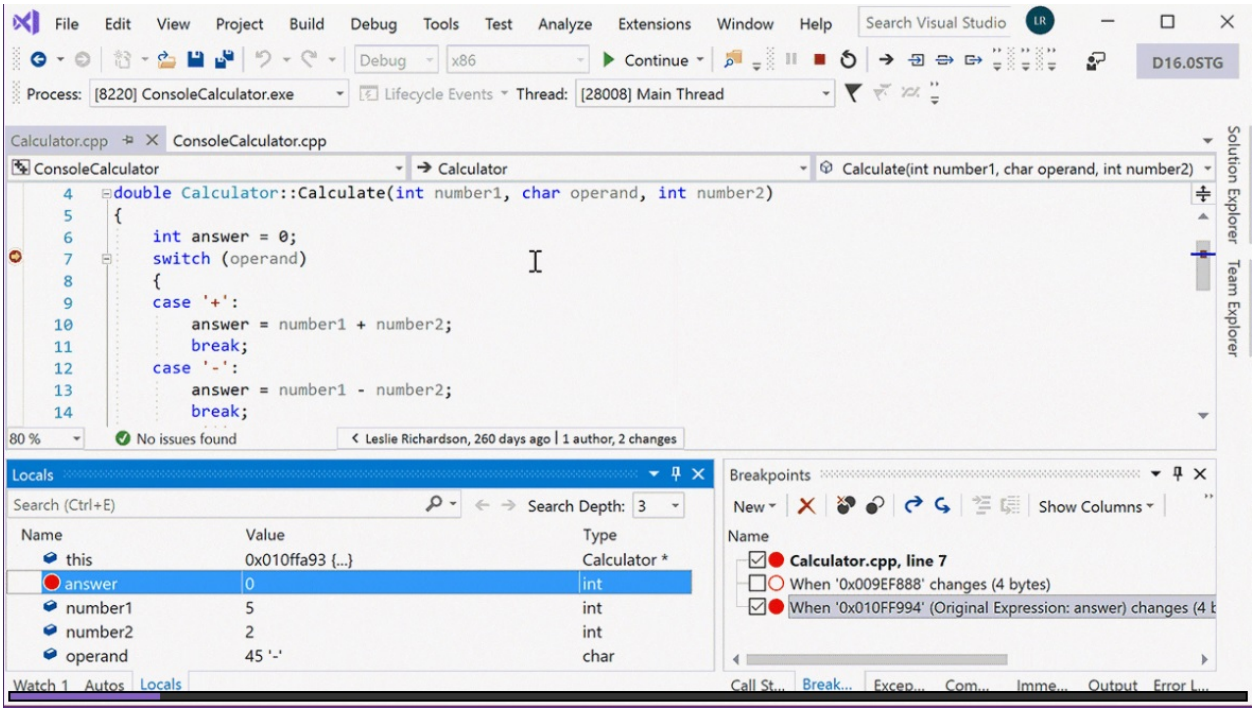
Spyder

Spyder is a Python IDE (https://github.com/spyder-ide/spyder) which is available with the Anaconda installation, so you should have it installed on your computer. It can be accessed from the Start Program menu, as shown below. Spyder offers most of the common IDE features. An interesting feature that is not offered by many similar IDEs is the variable explorer, which displays the user data and variables in a table-based layout inside the IDE.



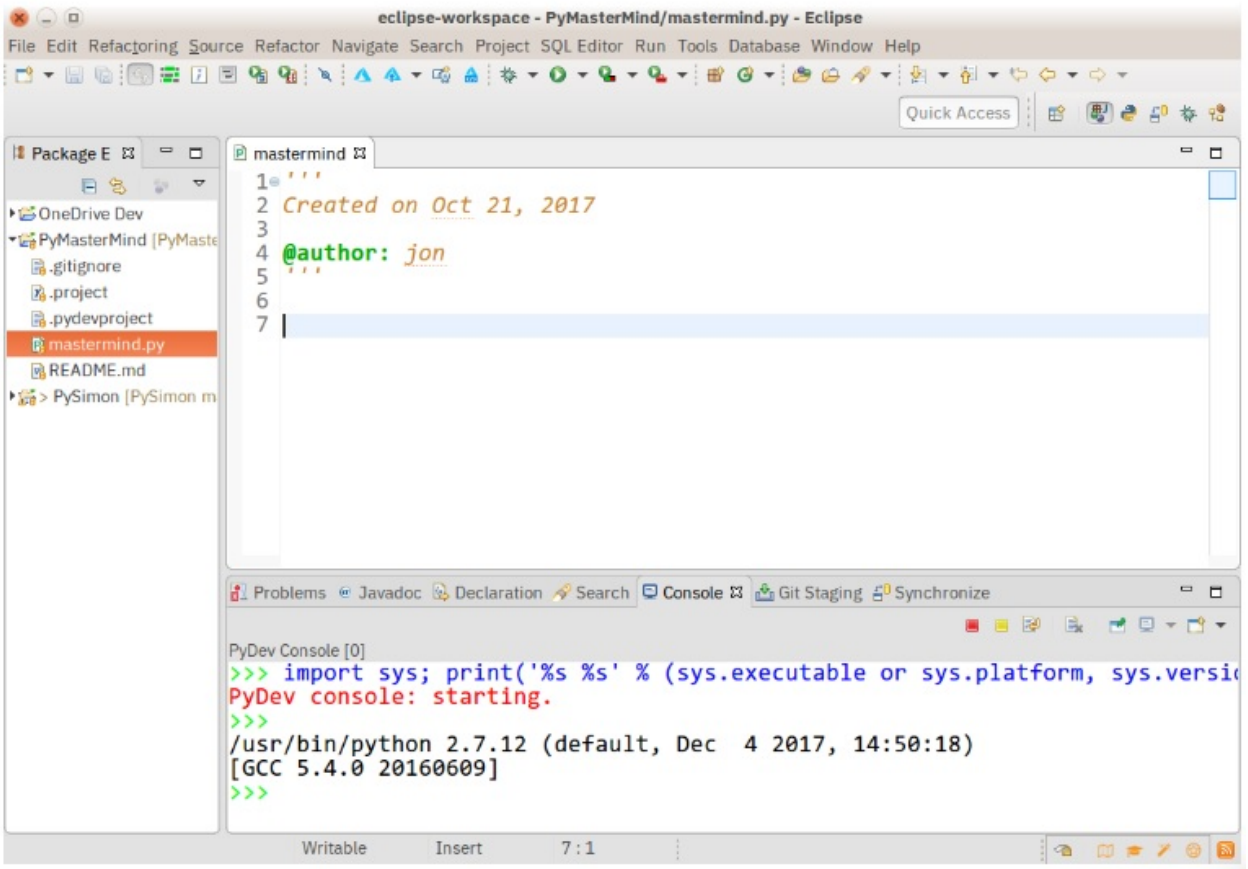
Visual Studio

Visual Studio is a general-purpose IDE (https://visualstudio.microsoft.com/vs/) developed by Microsoft. It is available for Windows and Mac OS, both as free (Community) and paid (Professional and Enterprise) versions. Visual Studio provides support for program development in different programming languages. Python Tools for Visual Studio (PTVS) enables Python coding in Visual Studio. VS is a large download, but it provides excellent tools for different languages.



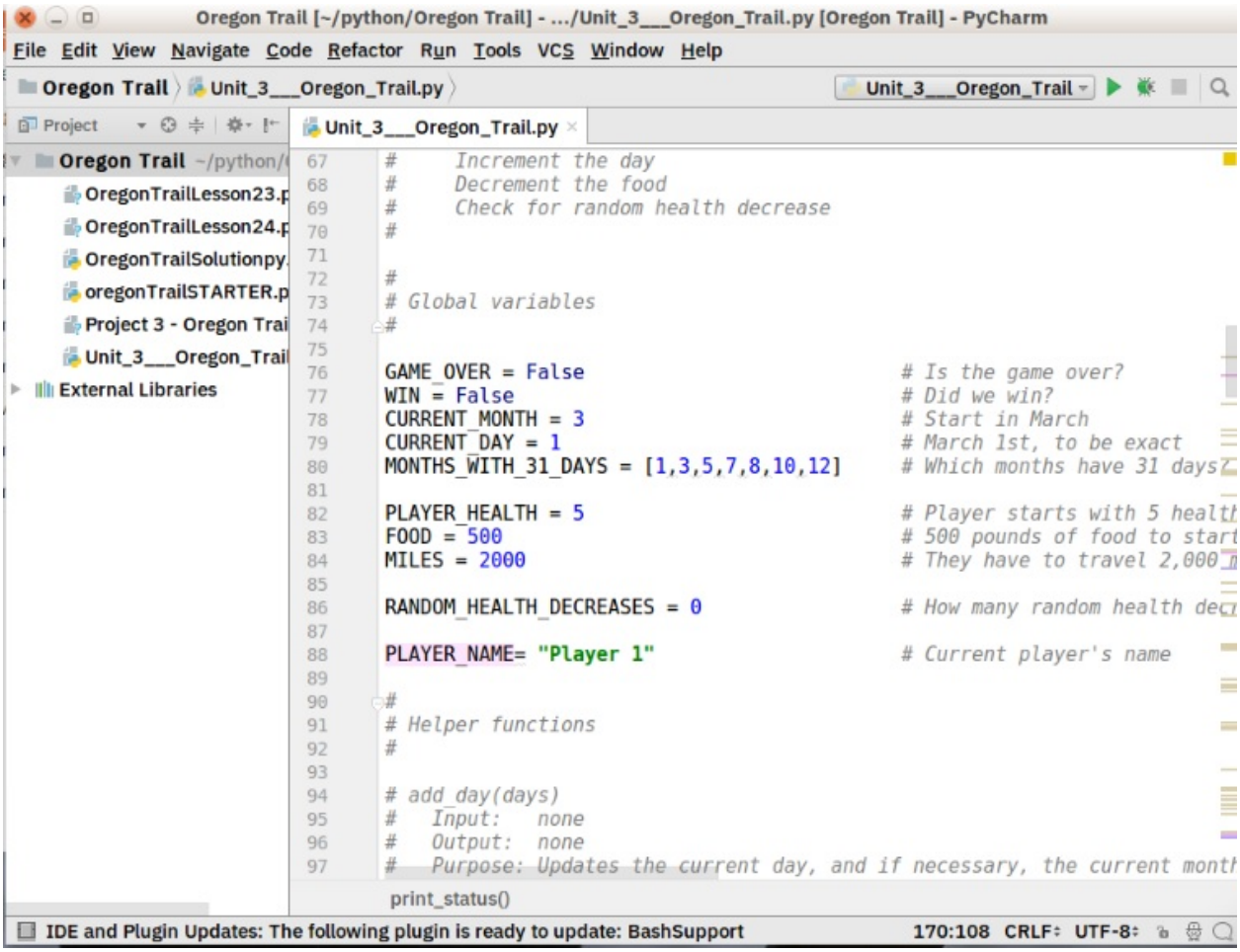
Eclipse and PyDev

Eclipse and PyDev is an IDE (www.eclipse.org), originally developed as a Java IDE, but it supports Python development by installing the PyDev plug-in. It is a popular and powerful IDE for Python development. It is available for Windows, Linux, and Mac OS. However, it may not be the easiest IDE for beginner programmers.



PyCharm

PyCharm is a fully dedicated IDE for Python (https://www.jetbrains.com/pycharm/). Available in both paid (Professional) and free open-source (Community) editions, PyCharm installs quickly and easily on Windows, Mac OS, and Linux platforms. PyCharm allows writing, running, and debugging Python code, and it has support for source control and projects.

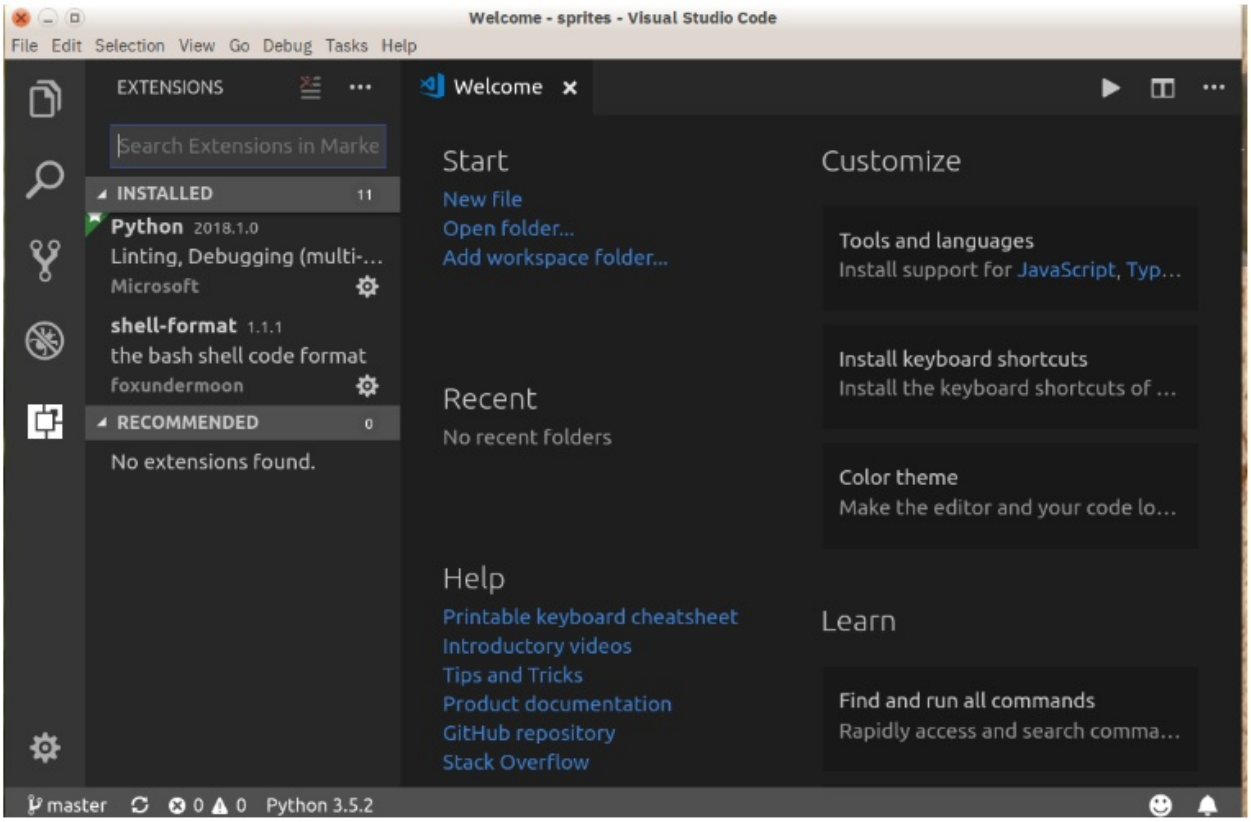


Other IDEs

Beside these, there are also other general-purpose IDEs and code editors, such as Komodo, Sublime Text, Atom, GNU Emacs, and other IDEs dedicated to Python code development, such as PythonWin, NetBeans,Thonny, and many others.

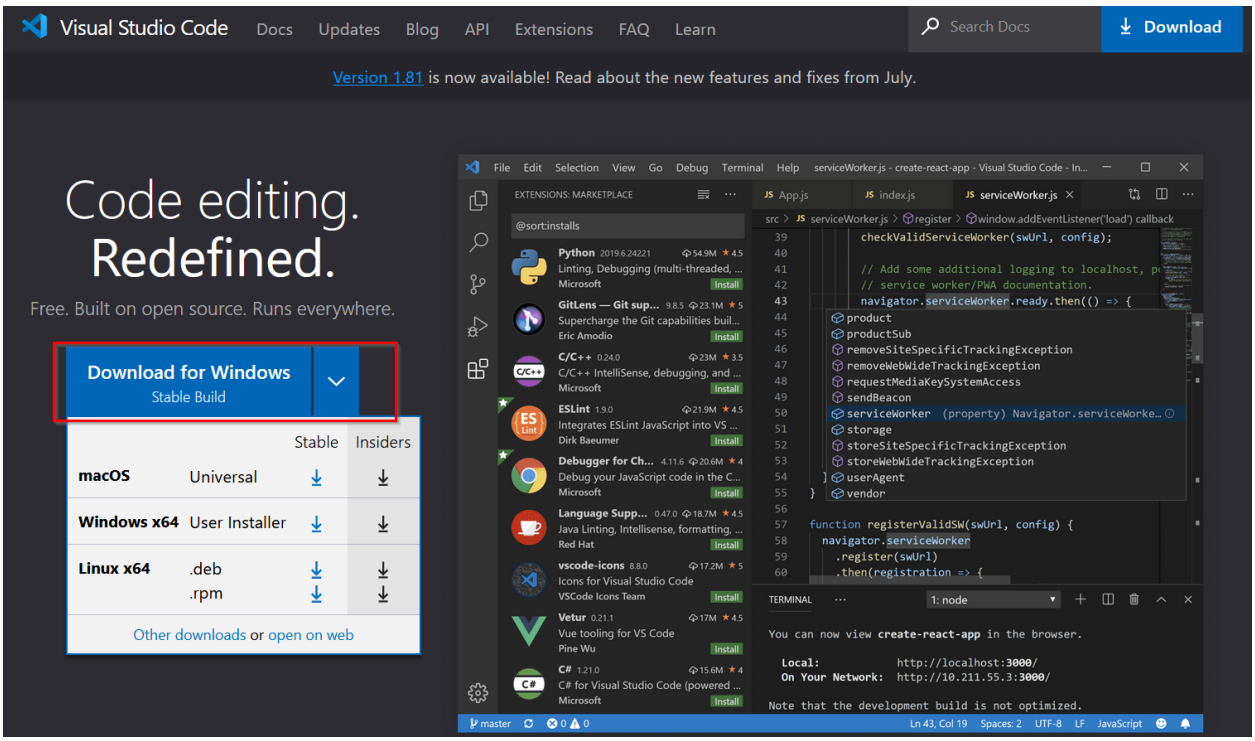
7.30.2 Introduction to VS Code

Visual Studio Code (VS Code) is a code editor (https://code.visualstudio.com/), available for Windows, Linux, and Mac OS. Differently from Visual Studio IDE listed above, Visual Studio Code is small and lightweight, open-source, and extensible. Installing Python support is easy, and VS Code will recognize your Python installation and libraries automatically. VS Code is built using Electron, a framework for creating desktop applications using JavaScript, HTML, and CSS. Visual Studio Code is recognized for its rapid launch speed and efficient resource utilization, guaranteeing a seamless coding experience, even with less powerful hardware.



VS Code Download and Installation

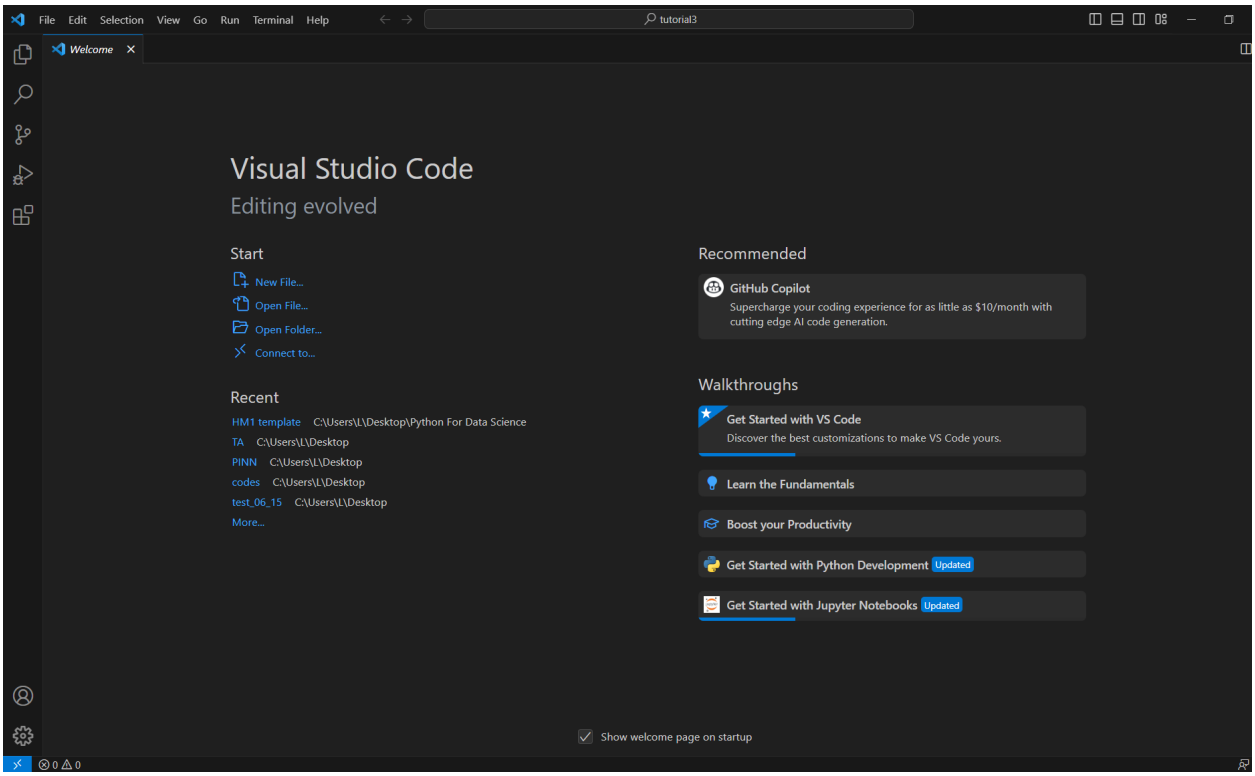
VS Code can be downloaded from: https://code.visualstudio.com/



During the installation, choose all the default options.

After the installation is completed, open the VS Code, and all initial settings can be selected as defaults.

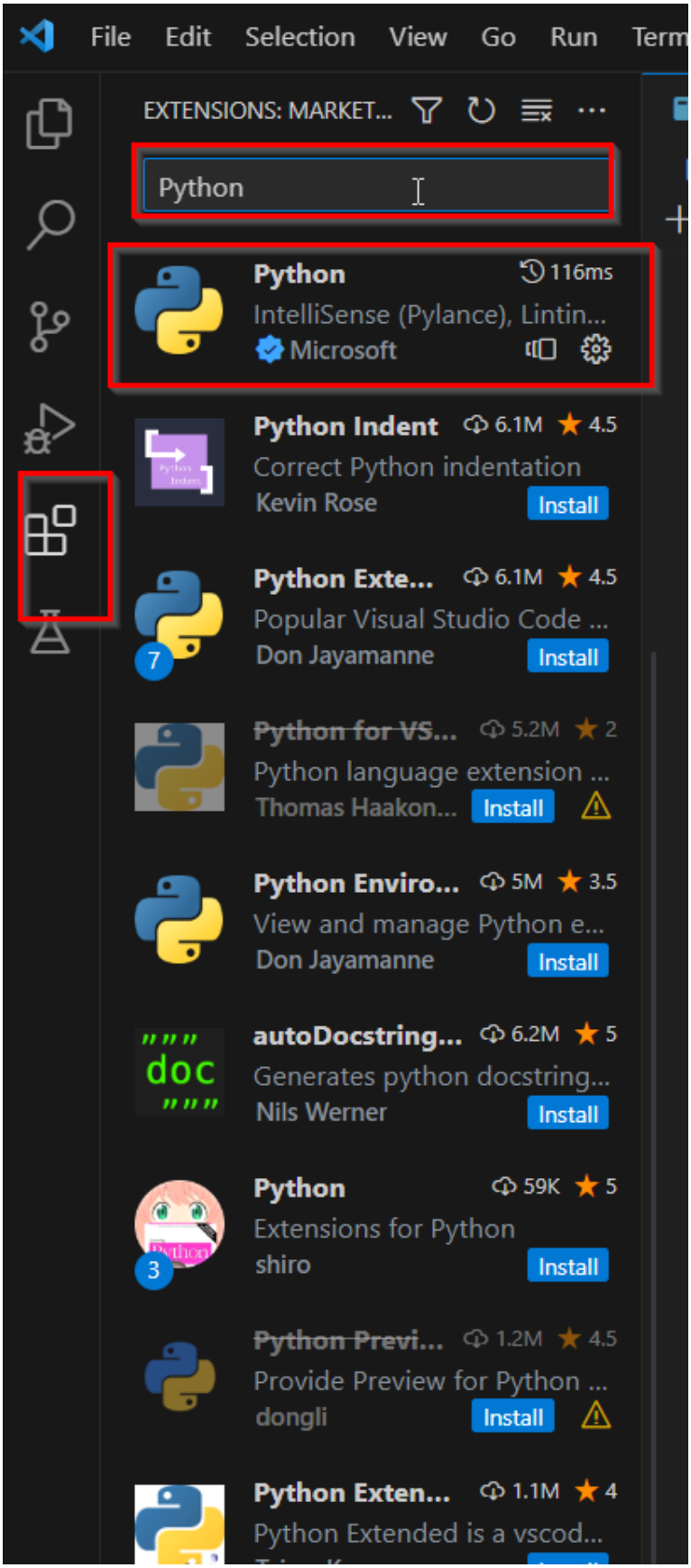
Then you will see the window similar to the following figure.



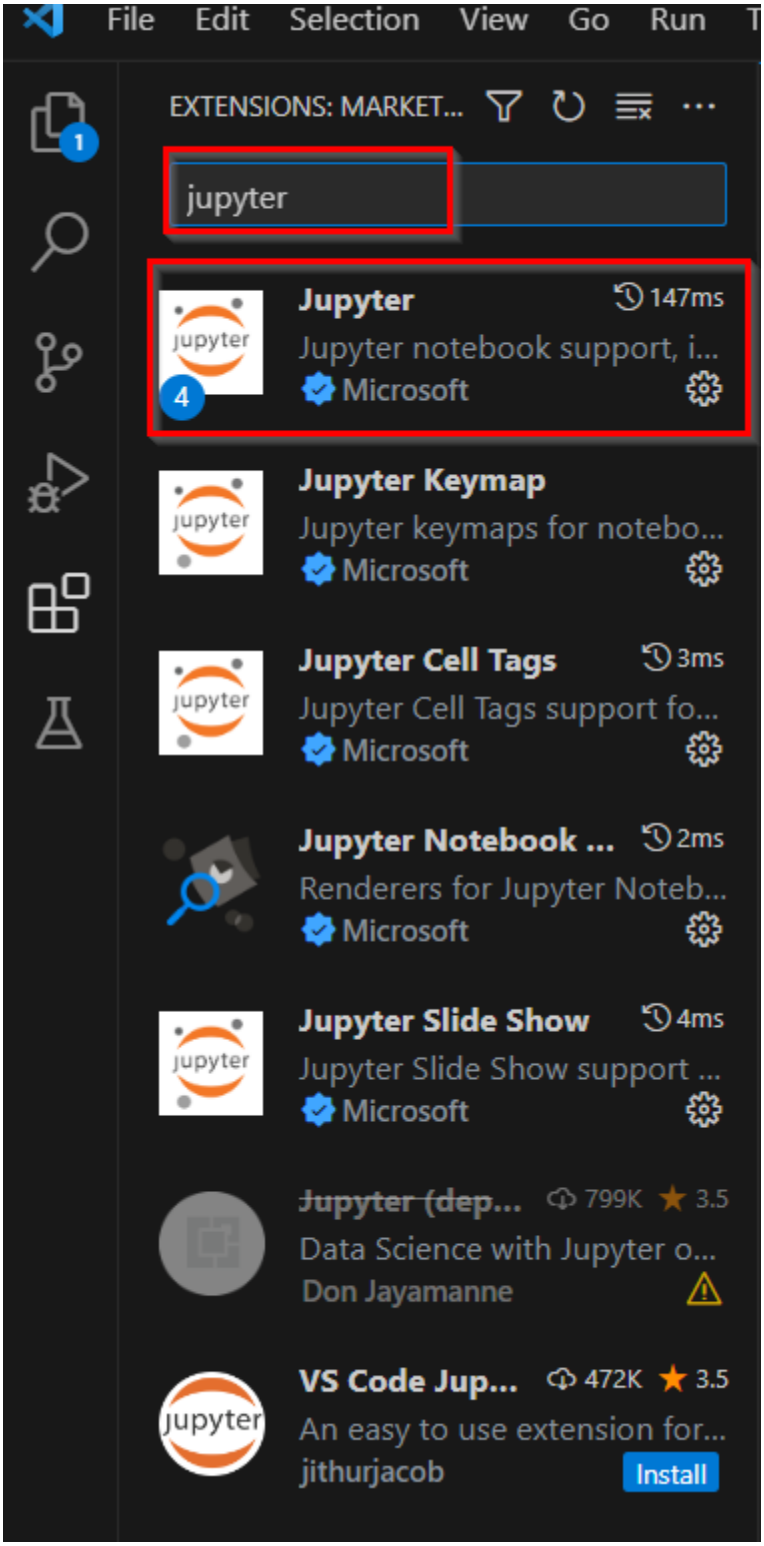
Python Extension Installation and Configuration

To run Python in VS Code, we need to install the Python extension and Jupyter extension first.

On the left sidebar of the VS Code Window, click the `Extension` button and install the Python Extension.


File Edit Selection View Go Run Term
EXTENSIONS: MARKET...
Python
Python
116ms
IntelliSense (Pylance), Lintin...
Microsoft
Python Indent
6.1M
4.5
Correct Python indentation
Kevin Rose
Install
Python Exte...
6.1M
4.5
Popular Visual Studio Code ...
Don Jayamanne
Install
Python for VS...
5.2M
2
Python language extension ...
Thomas Haakon...
Install
Python Enviro...
5M
3.5
View and manage Python e...
Don Jayamanne
Install
autoDocstring...
6.2M
5
Generates python docstring...
Nils Werner
Install
Python
59K
5
Extensions for Python
shiro
Install
Python Previ...
1.2M
4.5
Provide Preview for Python ...
dongli
Install
Python Exten...
1.1M
4
Python Extended is a vscod...

Similarly, search for the Jupyter notebook extension and install the extension.

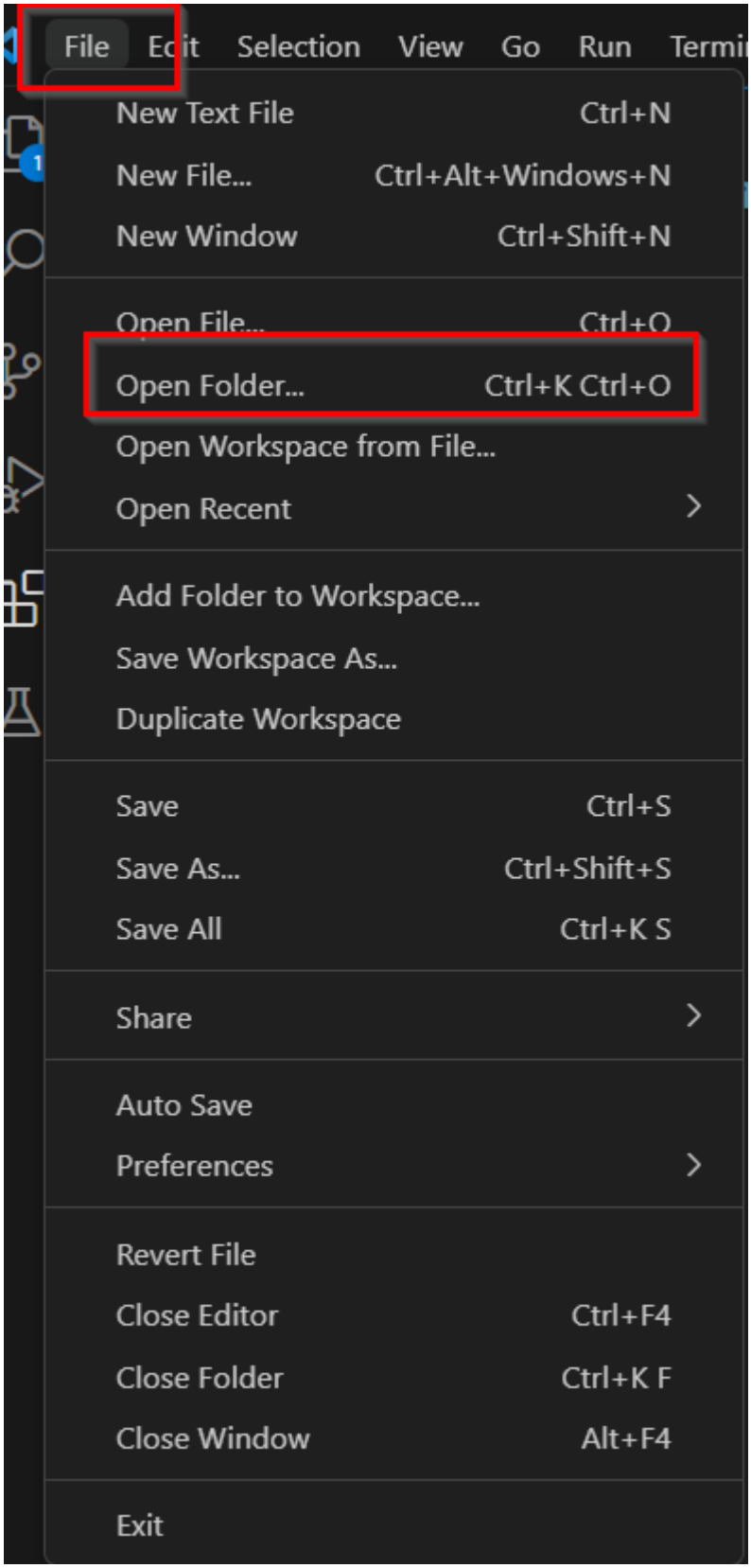


Start Jupyter Notebook in VS Code

To start a project, it is preferred to create a new folder for the project and store all files in that specific folder.

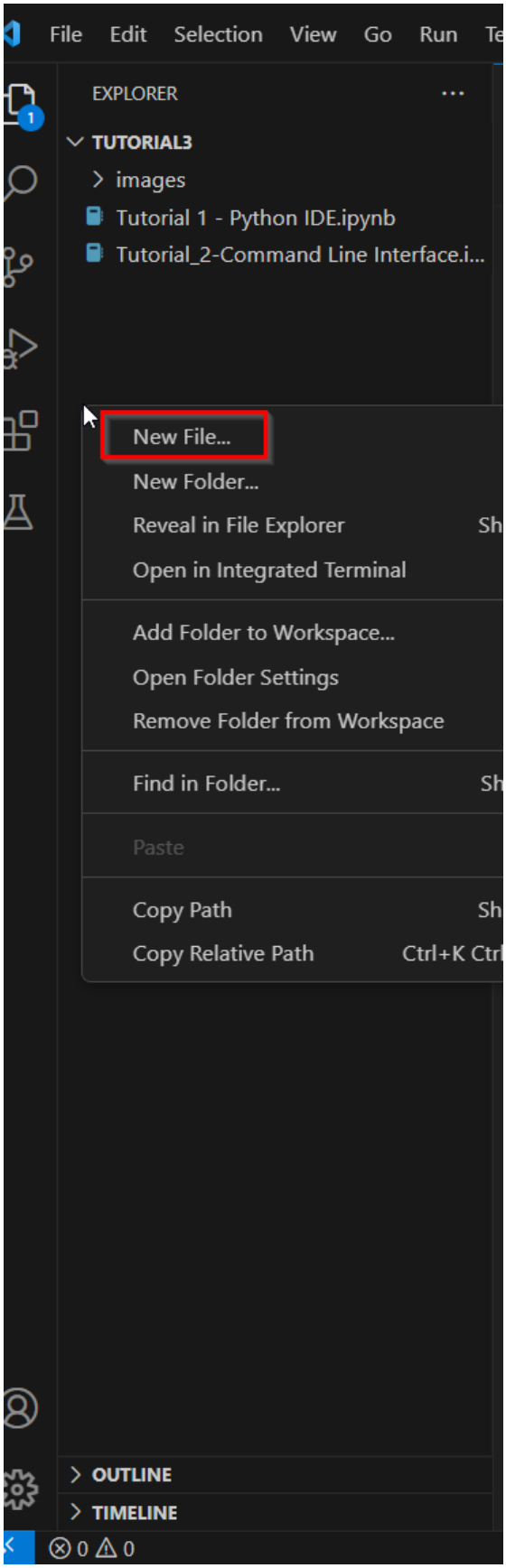
Then, we can open the folder in VS Code and create files/scripts inside the folder.

1) Click `File` and `Open Folder` in the upper left corner of the window to open the folder that you want to start your project in.



2. There are two ways to create a new Jupyter Notebook file inside a folder.

- Click `File` and then click `New File`, afterward choose the `Jupyter Notebook` option.
- Right-click in the blank area under your folder's name on the left side of the window to create a `New File`, then type your file name with `.ipynb`.

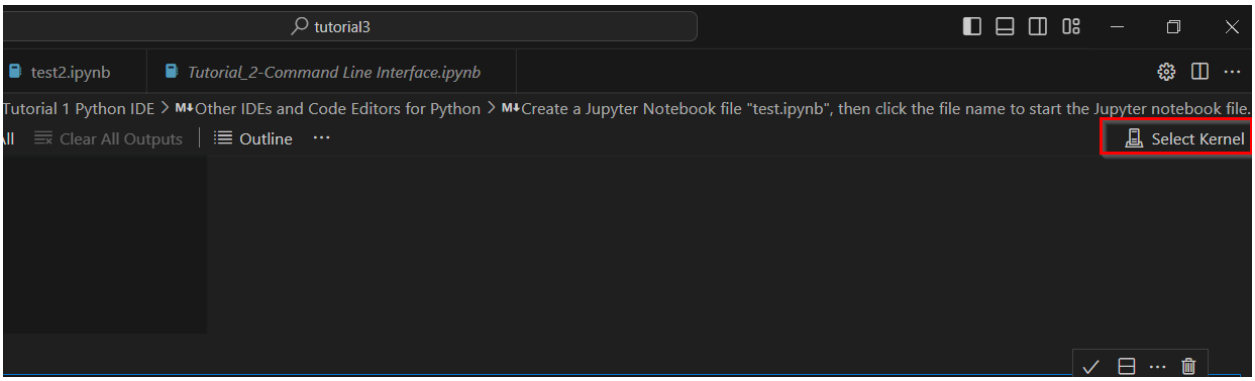

File
Edit
Selection
View
Go
Run
EXPLORER
TUTORIAL3
images
Tutorial 1 - Python IDE.ipynb
Tutorial_2-Command Line Interface.i...
New File...
New Folder...
Reveal in File Explorer
Open in Integrated Terminal
Add Folder to Workspace...
Open Folder Settings
Remove Folder from Workspace
Find in Folder...
Paste
Copy Path
Copy Relative Path
Ctrl+K Ctrl
OUTLINE
TIMELINE
0 0

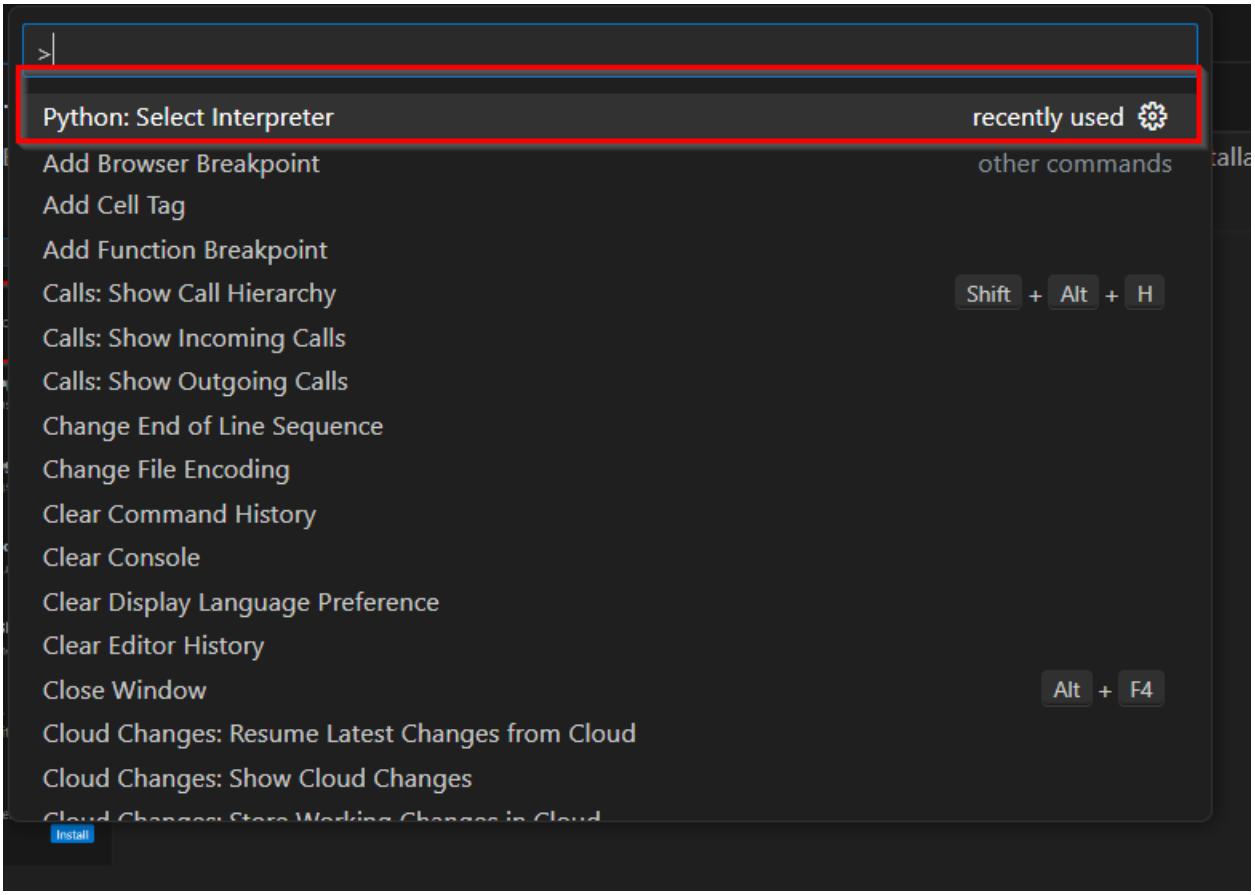
3) The first time we run a new file, we need to manually select the kernel.

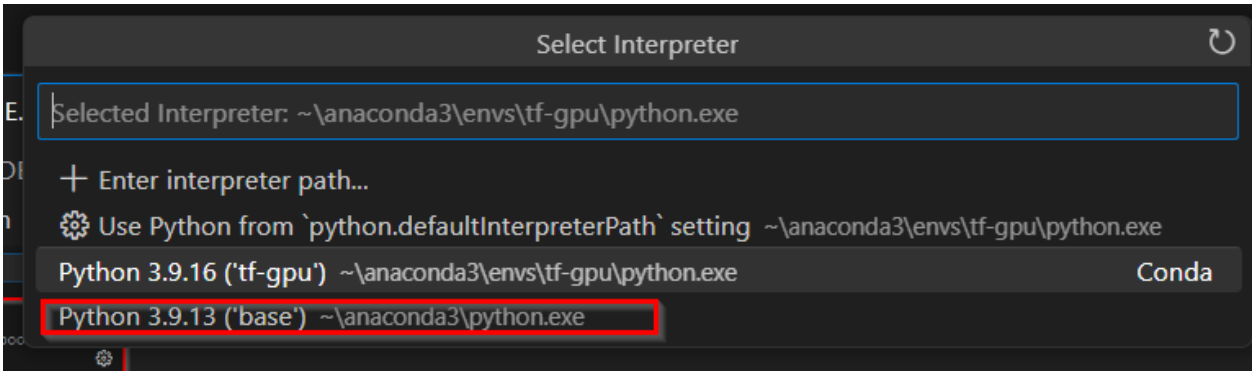
   In the upper right corner of the screen, click `Select Kernel` to select the Python interpreter.

   We can select the default Python interpreter from our previous Anaconda installation.

   With that, the installation of Python in VS Code is completed. If any extensions are needed in the future when you run some specific scripts, VS Code will automatically show a pop-up window to help you with the installation.



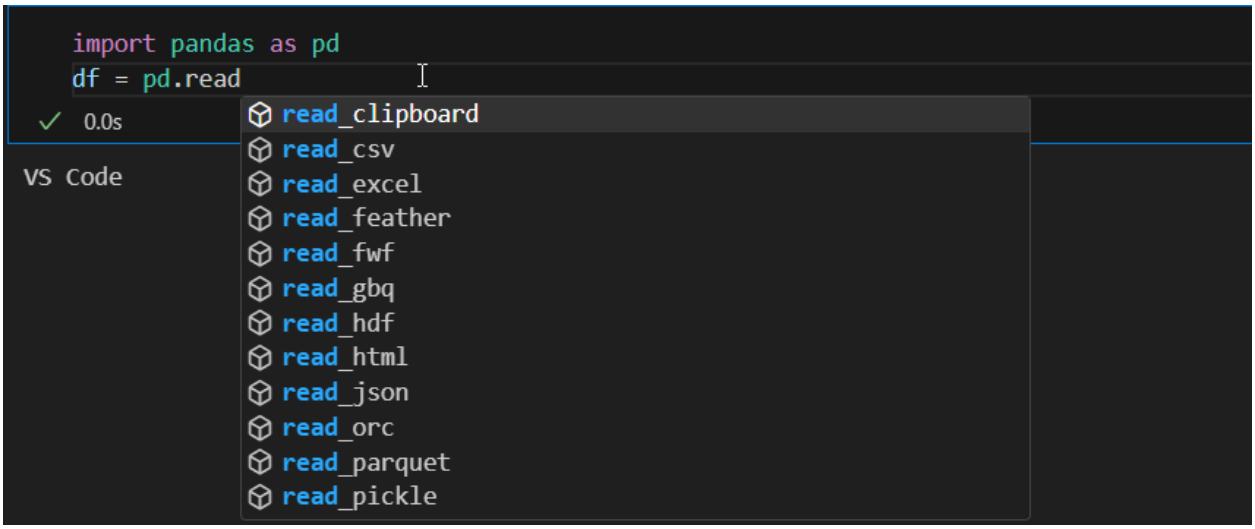




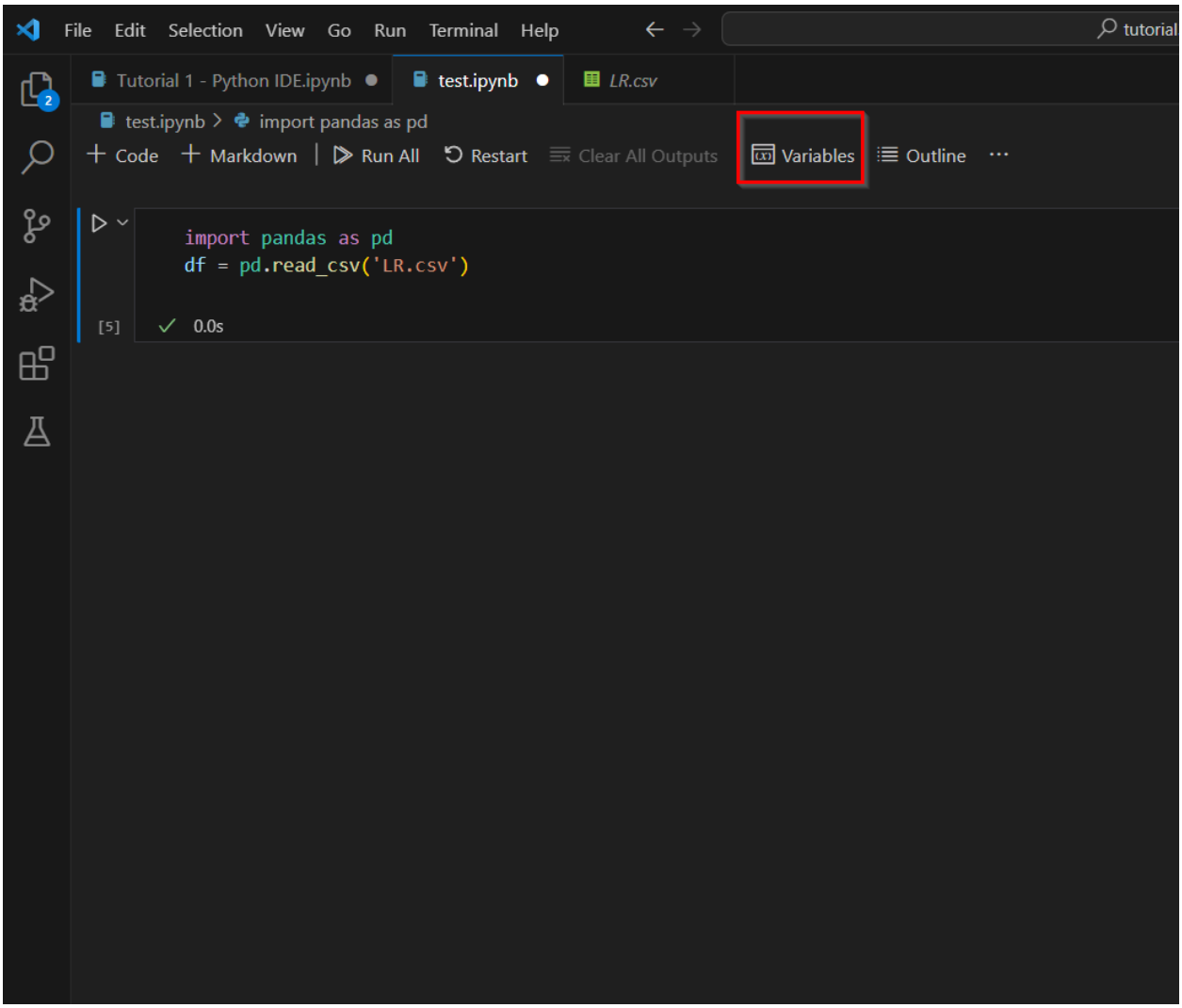
VS Code features

The built-in IntelliSense feature provides key functions such as:

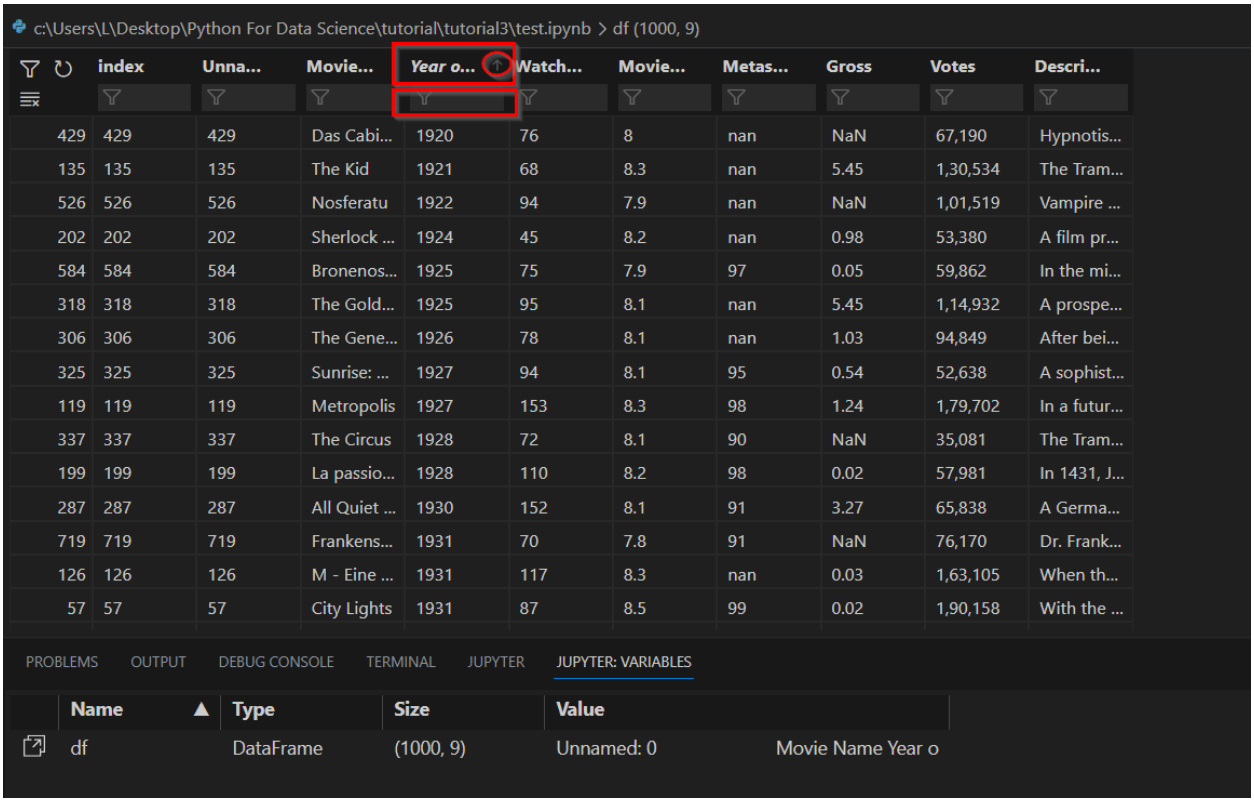
- Auto code completion/Assist/Hint: It predicts and auto-generates code as you type, reducing the requirement for manual input and diminishing the chances of typos and syntax errors. It offers recommendations for code elements that align with the current context, simplifying the process of producing accurate code. It also considers the programming language in use and the libraries or modules that you've incorporated. This results in suggestions that are closely aligned with your code's context, facilitating the exploration of accessible choices.
- Parameter Information: When you call a function/method, it shows information about the function's parameters, the data types, and descriptions. This makes it simpler to provide the correct arguments when calling functions.



Variables/Data Viewer enables to examine and engage with the variables and data within your Jupyter notebook's kernel. It provides a visual interface for investigating the present condition of variables, their values, and data arrangements.

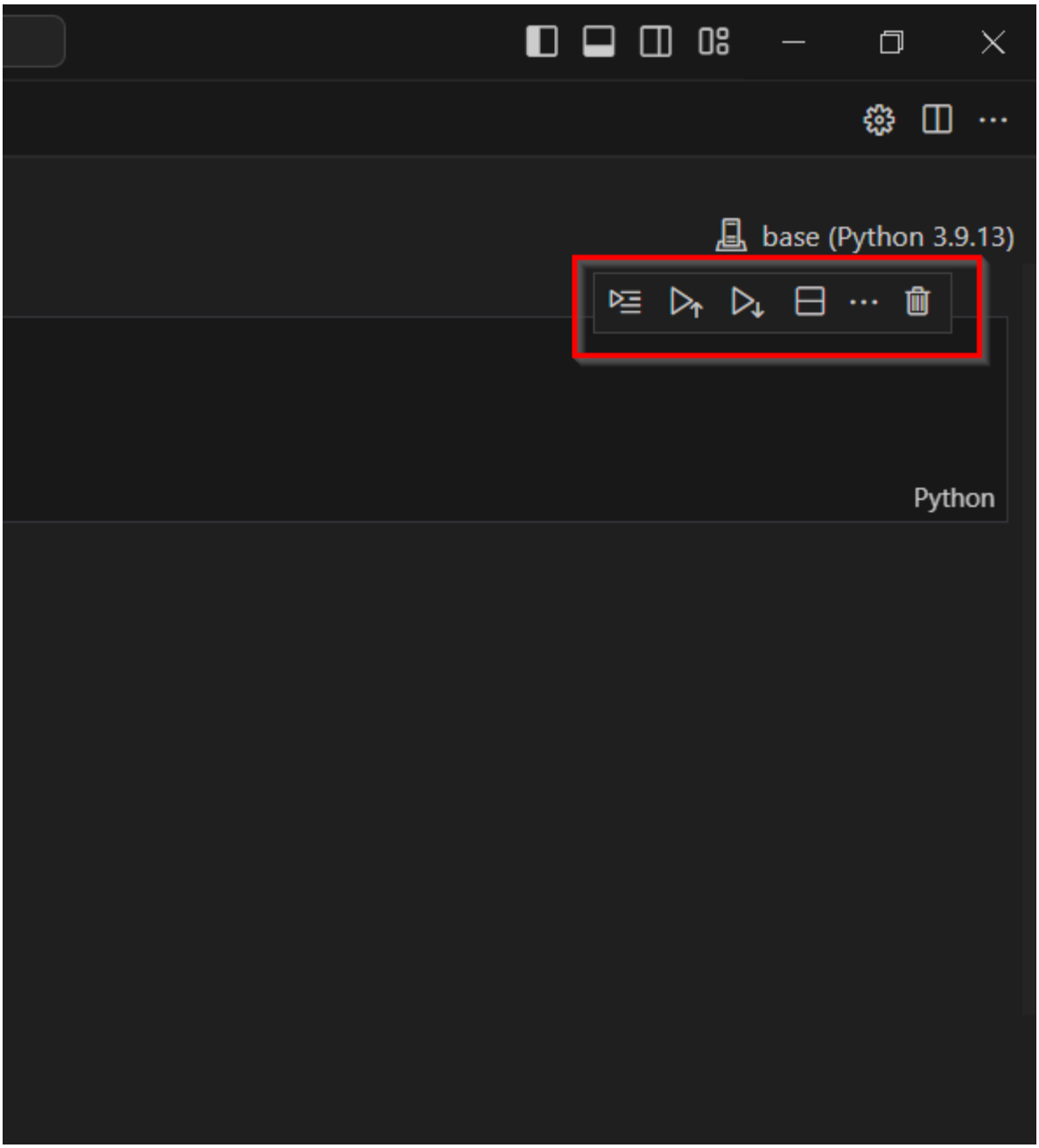


In the Data Viewer, you can `sort` the columns by clicking on the column name, and you can also `filter` the data by typing the keywords/syntax.

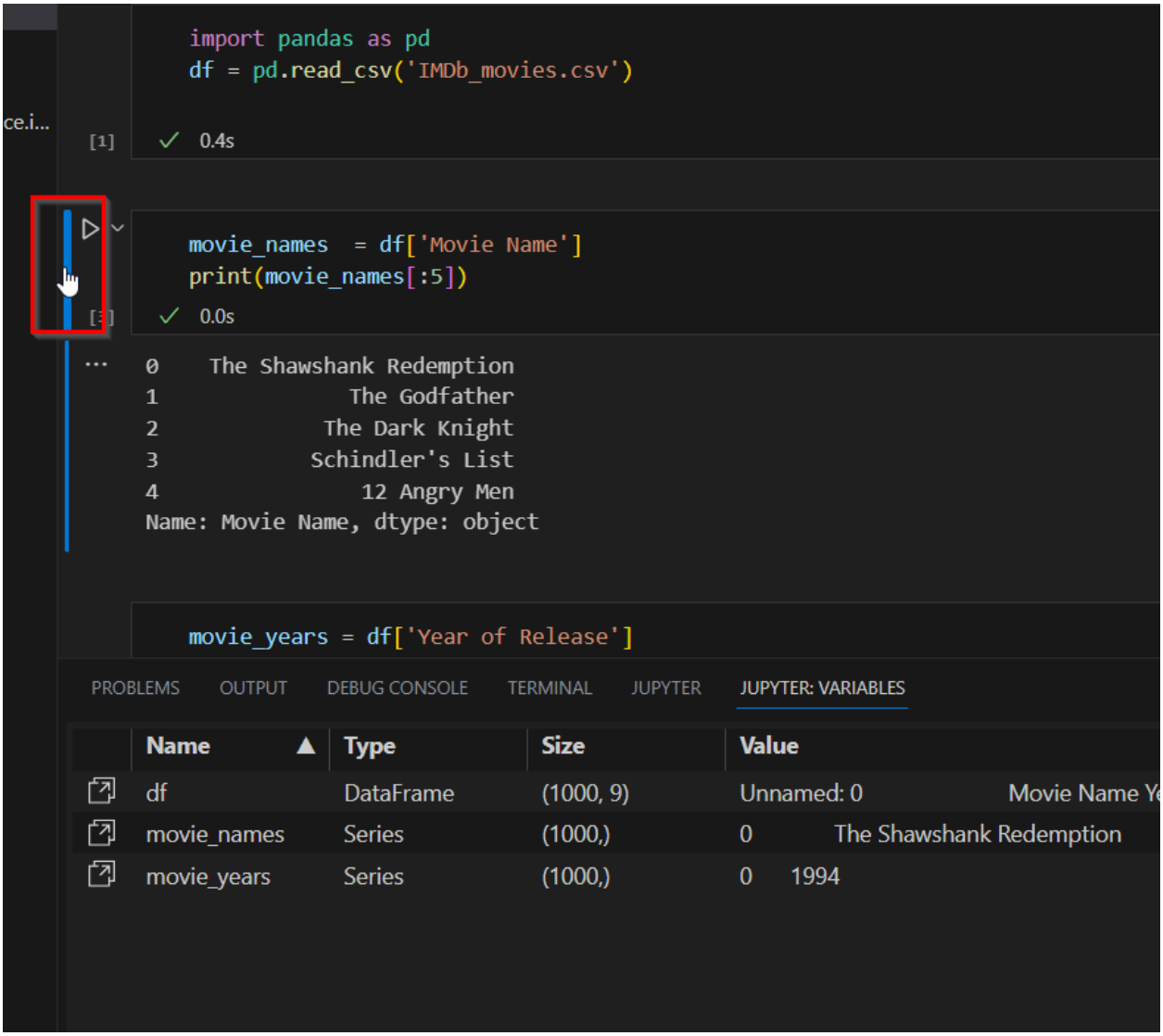


Run Cell Operations include:

- Run by Line, which runs the code line by line.
- Execute Above Cells, which runs all the cells above the current cell.
- Execute Cell and Below Cells, which runs the current and all the below cells.



To move cells, click and hold the blue bar in front of the cell, then move to any place you prefer.

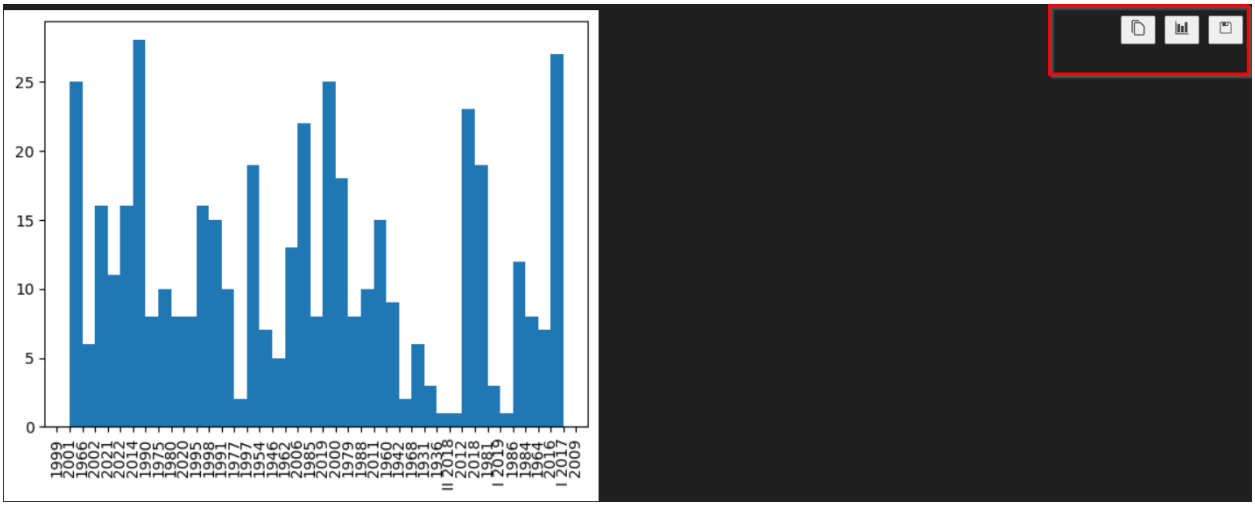


Copy/Save/Zoom-in Plots.

Copy function allows you to copy the plot to your clipboard.

Save function allows you the save the plot as an image file.

Zoom-in allows you to zoom in and see the details if the plot is too large and complicated.



Debug features: click the arrow next to the Run Cell button and click `Debug Cell`, and you can then debug like you normally do in Python.

BACK TO TOP

7.31 Tutorial 4 - Virtual Environments

A **virtual environment** is a tool that helps to keep dependencies required by different projects separate by creating isolated Python virtual environments for them. Virtual environments create containers for each project, so that the Python interpreter, libraries, and scripts installed in one virtual environment are isolated from those installed in other virtual environments, as well as they are isolated from the libraries and packages that are installed in the main Python installation.

Therefore, where we create and activate a virtual environment for a specific project, the project becomes an independent application, which is isolated from the system-installed Python and from all other Python libraries. This project-specific virtual environment provides its own Python interpreter, and its own pip to install libraries, separated from other Python libraries.

By creating and managing separate environments for different projects, there is no chance of breaking currently installed packages in other environments. It also helps with reproducibility among developers and researchers working on large projects. For instance, if you have several different projects with their own virtual environments, they can have different versions of a library: for example, one project can use TensorFlow 1.12, and another can use TensorFlow 2.5. This way, we won't worry whether an update to the TensorFlow library in the main system-installed Python would impact the code in all our projects.

There are several tools for managing virtual environments. Python 3.3 provides a standard module `venv`, which is most commonly used for managing virtual environments. It allows to manage separate package versions for different projects. When creating a new project, we can simply create a new virtual environment.

- The full official documentation for `venv` can be found here
- The full official user guide for `venv` can be found here
- The PEP proposal for `venv` can be found here

If you are looking for practical examples, it is recommended to consult the user guide. However, if you are looking for more information about specific details of `venv`, consulting the full documentation is recommended.

Installing venv

The module `venv` should be included in the standard Python library. If for some reason you must install it, this can be done with the following command:

`python3 -m pip install --user virutalenv`

Note: on Debian/Ubuntu systems, you will need to install the python3-venv package using the following command:

`apt install python3.10-venv`.

7.31.1 Creating a Virtual Environment

To create a virtual environment, run the following command:

To create a virutal envonment, run the following command:

`python3 -m venv path/to/new/virutal/environment` (in Unix/MacOS)

`py -m pip install --user <virtual_environment_name>` (in Windows)

Alternatively, there is a script installed with the `venv` library to make this more slightly more convenient:

`pyvenv /path/to/new/virtual/environment` (in Unix/MacOS)

`python -m <virtual_environment_name> ./venv` (in Windows)

Activating a Virtual Environment

Before we can start installing packages in the virtual environment, we must activate it. Doing so will put the virtual environment-specific Python and pip executables in your shell's `PATH`.

To activate a virtual environment, run the following command:

`source <path-to-venv>/bin/activate` (in Unix/MacOS)

`.\<virtual_environment_name>\Scripts\activate` (in Windows)

To confirm that the virtual environment has been activated you can check the location of your Pytton interpreter:

`which python` (in Unix/MacOS)

`where python` (in Windows)

As long as the environment is active, you'll be able to import packages installed in the environment.

To leave the environment run:

`deactivate`

Installing Packages

Make sure that the environment you wish to install packages into is active. Once this is done, it is as simple as installing packages through `pip` as you would normally. An example command to install `requests` (a popular library for making HTTP requests) is:

`python3 -m pip install requests` (in Unix/MacOS)

`pip install requests` (in Windows)

To check the list of all packages installed in the newly created virtual environment, use:

```
pip list
```

Similarly, we can generate a text file listing all installed libraries in a virtual environment with:

```
pip freeze > requirements.txt
```

This can be convenient, because if other users would like to replicate your virtual environment, instead of installing all libraries one by one, they can just run:

```
pip install -r requirements.txt
```

A Word of Caution and Some Best Practices

When a script is installed using `venv`, its shebang line (starting with `#!`) points to the environment's Python interpreter. This means that these environments are inherently non-portable.

Moreover, if one is careful to include the **absolute path** to the desired environment interpreter in the shebang line, it is not necessary to activate the virtual environment before running the script.

For example, if you have an environment called `env` that has some non-standard packages installed, and you start writing a script that relies upon these packages but you did not want to (or can not) ensure the environment was active before running it, simply put something like the following as the shebang line:

```
#!/<virutal_environment_name>/bin/python
```

7.31.2 Conda Environment

`Conda Env` is a self-contained and isolated workspace within the Conda package management system, similar to `venv`. It allows to create and manage specific environments for different projects or applications, each with its own set of packages and dependencies.

The full official documentation for `Conda Env` can be found here.

`Conda Env` is pre-installed with Anaconda.

Creating a `Conda Env` with Commands

1) Open the terminal or an Anaconda Prompt.
2) Type: `conda create --name myenv` (or you can specify the python version: `conda create -n myenv python=3.9`).
3) Type `y` to proceed.

To activate a `Conda Env`, type:

```
conda activate myenv
```

To list all installed conda environments, type:

`conda env list`

To activate the base conda environment from the command prompt in Windows, just type:

`conda activate`

To deactivate the `Conda Env` you are in, type:

`conda deactivate`

Install Packages in a Conda Env

To install `scikit-learn`, for example, using Conda commands, type:

`conda install -c anaconda scikit-learn`

Or, you can use pip install as in:

`pip install -U scikit-learn`

BACK TO TOP

7.32 Tutorial 5 - Web Scraping

This tutorial is adapted from the original tutorial Guide to Web Scraping in the *Complete Python 3 Bootcamp* by Pierian Data.

Before we begin with web scraping and Python, here are some important rules to follow and understand:

1. Always be respectful and try to get permission to scrape. Do not bombard a website with scraping requests, otherwise your IP address may be blocked.
2. Be aware that websites change often, meaning your code could go from working to totally broken from one day to the next.
3. Pretty much every web scraping project is a unique and custom job, so try your best to generalize the skills learned here.

7.32.1 Basic components of a WebSite

HTML

HTML stands for Hypertext Markup Language and every website on the Internet uses it to display information. Even the Jupyter Notebook system uses it to display this information in your browser. If you right-click on a website and select "View Page Source" you can see the raw HTML of a web page. This is the information that Python will be looking at to grab information from. Let's take a look at the HTML organization of a simple webpage:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Title on Browser Tab</title>
    </head>
    <body>
```

(continues on next page)

(continued from previous page)

```
        <h1> Website Header </h1>
        <p> Some Paragraph </p>
    </body>
</html>
```

Let's break down these components.

Every <tag> indicates a specific block type on the webpage:

1. The <!DOCTYPE html> is the tag for type declaration that HTML documents always start with this, letting the browser know it is an HTML file.
2. The component blocks of the HTML document are placed between <html> and </html>.
3. Metadata and script connections (like a link to a CSS file or a JS file) are often placed in the <head> block.
4. The <title> tag block defines the title of the webpage (this is what shows up in the tab of a website you're visiting).
5. Between <body> and </body> tags are the blocks that will be visible to the site visitor.
6. Headings are defined by the <h1> through <h6> tags, where the number represents the size of the heading.
7. Paragraphs are defined by the <p> tag, and this is essentially just normal text on the website.

There are many more tags than these, such as <a> for hyperlinks, <table> for tables, <tr> for table rows, <td> for table columns, and more.

CSS

CSS stands for Cascading Style Sheets, and it is what gives "style" to a website, including colors and fonts, and even some animations. CSS uses tags such as <id> or <class> to connect an HTML element to a CSS feature, such as a particular color. The **id** tag is a unique id for an HTML tag and must be unique within the HTML document. The **class** tag defines a general style that can be linked to multiple HTML tags. Basically if you only want a single html tag to be read, you would use an id tag, and if you wanted several HTML tags/blocks to be read, you would create a class in your CSS doc, and then link it to the rest of these blocks.

7.32.2 Web Scraping with Python

Keep in mind again that you should always have permission for the website you are scraping. Check a website's terms and conditions for more info. Also keep in mind that a computer can send requests to a website very fast, so a website may block your computer's IP address if you send too many requests too quickly.

There are a few libraries you will need for this task, which you can install with `conda install` if you are using Anaconda distribution.

```
conda install requests
conda install lxml
conda install bs4
```

If you are not using Anaconda distribution, you can use `pip install`.

```
pip install requests
pip install lxml
pip install bs4
```

[1]:
```
# Install libraries
!pip install requests
!pip install lxml
!pip install bs4
```

```
Requirement already satisfied: requests in c:\users\vakanski\anaconda3\lib\site-packages␣
↪(2.31.0)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\vakanski\anaconda3\
↪lib\site-packages (from requests) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in c:\users\vakanski\anaconda3\lib\site-
↪packages (from requests) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\vakanski\anaconda3\lib\
↪site-packages (from requests) (1.26.16)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\vakanski\anaconda3\lib\
↪site-packages (from requests) (2023.7.22)
Requirement already satisfied: lxml in c:\users\vakanski\anaconda3\lib\site-packages (4.
↪9.2)
Collecting bs4
  Downloading bs4-0.0.1.tar.gz (1.1 kB)
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'done'
Requirement already satisfied: beautifulsoup4 in c:\users\vakanski\anaconda3\lib\site-
↪packages (from bs4) (4.12.2)
Requirement already satisfied: soupsieve>1.2 in c:\users\vakanski\anaconda3\lib\site-
↪packages (from beautifulsoup4->bs4) (2.4)
Building wheels for collected packages: bs4
  Building wheel for bs4 (setup.py): started
  Building wheel for bs4 (setup.py): finished with status 'done'
  Created wheel for bs4: filename=bs4-0.0.1-py3-none-any.whl size=1264␣
↪sha256=3547454165022 0b5e135d4bc9cb76cfa5caba3ae35e2c45fe611f4ecb52838ec
  Stored in directory: c:\users\vakanski\appdata\local\pip\cache\wheels\d4\c8\5b\
↪b5be9c20e5e4503d04a6eac8a3cd5c2393505c29f02bea0960
Successfully built bs4
Installing collected packages: bs4
Successfully installed bs4-0.0.1
```

Example Task 0 - Grabbing the Title of a Page

Let's start very simple and grab the title of a page. Remember that this is the HTML block with the **title** tag. For this task, we will use **www.example.com** which is a website specifically made to serve as an example domain. Let's go through the main steps:

[2]:
```
import requests
```

[3]:
```
# Step 1: Use the requests library to grab the page
# Note, this may fail if you have a firewall blocking Python/Jupyter
# Note that sometimes you need to run this twice if it fails the first time
res = requests.get("http://www.example.com")
```

The type of the object `res` is a `requests.models.Response` object, and it actually contains the information from the website as shown in the following cell.

[4]:
```
type(res)
```

[4]:
```
requests.models.Response
```

[5]:
```
res.text
```

[5]:
```
'<!doctype html>\n<html>\n<head>\n    <title>Example Domain</title>\n\n    <meta charset=
↪"utf-8" />\n    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />\
↪n    <meta name="viewport" content="width=device-width, initial-scale=1" />\n
↪<style type="text/css">\n    body {\n        background-color: #f0f0f2;\n        ␣
↪margin: 0;\n        padding: 0;\n        font-family: -apple-system, system-ui,␣
↪BlinkMacSystemFont, "Segoe UI", "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-
↪serif;\n        \n    }\n    div {\n        width: 600px;\n        margin: 5em auto;\n␣
↪        padding: 2em;\n        background-color: #fdfdff;\n        border-radius: 0.5em;
↪\n        box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);\n    }\n    a:link, a:visited
↪{\n        color: #38488f;\n        text-decoration: none;\n    }\n    @media (max-
↪width: 700px) {\n        div {\n            margin: 0 auto;\n            width: auto;\
↪n        }\n    }\n    </style>    \n</head>\n\n<body>\n<div>\n    <h1>Example Domain</
↪h1>\n    <p>This domain is for use in illustrative examples in documents. You may use␣
↪this\n    domain in literature without prior coordination or asking for permission.</p>
↪\n    <p><a href="https://www.iana.org/domains/example">More information...</a></p>\n</
↪div>\n</body>\n</html>\n'
```

To analyze the infromation extracted from the webpage, we use the library `BeautifulSoup`. Technically, we could use our own custom script to loook for items in the string of `res.text`, however the `BeautifulSoup` library already has lots of built-in tools and methods to grab information from a string of an HTML file. Using `BeautifulSoup` we can create a "soup" object that contains all the "ingredients" of the webpage.

[6]:
```
import bs4
```

[7]:
```
soup = bs4.BeautifulSoup(res.text)
```

[8]:
```
soup
```

[8]:
```
<!DOCTYPE html>
<html>
<head>
<title>Example Domain</title>
<meta charset="utf-8"/>
<meta content="text/html; charset=utf-8" http-equiv="Content-type"/>
<meta content="width=device-width, initial-scale=1" name="viewport"/>
<style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "Open Sans
↪", "Helvetica Neue", Helvetica, Arial, sans-serif;

    }
    div {
        width: 600px;
        margin: 5em auto;
        padding: 2em;
```

(continues on next page)

(continued from previous page)

```
        background-color: #fdfdff;
        border-radius: 0.5em;
        box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);
    }
    a:link, a:visited {
        color: #38488f;
        text-decoration: none;
    }
    @media (max-width: 700px) {
        div {
            margin: 0 auto;
            width: auto;
        }
    }
    </style>
</head>
<body>
<div>
<h1>Example Domain</h1>
<p>This domain is for use in illustrative examples in documents. You may use this
    domain in literature without prior coordination or asking for permission.</p>
<p><a href="https://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
```

Now let's use the `.select()` method to grab elements. Since we are looking for the `'title'` tag, so we will pass in `'title'`.

[9]:
```
soup.select('title')
```

[9]:
```
[<title>Example Domain</title>]
```

Notice that the returned object is actually a list containing all the title along with the tags. Since this object it still a specialized tag, we can use `getText()` to grab just the text.

[10]:
```
title_tag = soup.select('title')
```

[11]:
```
title_tag[0]
```

[11]:
```
<title>Example Domain</title>
```

[12]:
```
type(title_tag[0])
```

[12]:
```
bs4.element.Tag
```

[13]:
```
title_tag[0].getText()
```

[13]:
```
'Example Domain'
```

Example Task 2 - Grabbing All Elements of a Class

Let's try to grab all the section headings of the Wikipedia Article on Grace Hopper from this URL: https://en.wikipedia.org/wiki/Grace_Hopper

[14]:
```
# Get the request
res = requests.get('https://en.wikipedia.org/wiki/Grace_Hopper')
```

[15]:
```
# Create a soup from request
# The second parameter "lxml" is the type of parser to use.
# lxml is a faster alternative to python's native HTML parser.
soup = bs4.BeautifulSoup(res.text,"lxml")
```

To figure out what we are actually looking for, let's inspect the elements on the page.

Syntax to pass to the .select() method

Match Results

soup.select('div')

All elements with the tag

soup.select('#some_id')

The HTML element containing the id attribute of some_id

soup.select('.notice')

All the HTML elements with the CSS class named notice

soup.select('div span')

Any elements named that are within an element named

soup.select('div > span')

Any elements named that are directly within an element named , with no other element in between

The section headers have the class "mw-headline". Because this is a class and not a straight tag, we need to adhere to some syntax for CSS.

[17]:
```
# note that depending on your IP Address,
# this class may be called something different
soup.select(".mw-headline")
```

[17]:
```
[<span class="mw-headline" id="Early_life_and_education">Early life and education</span>,
 <span class="mw-headline" id="Career">Career</span>,
 <span class="mw-headline" id="World_War_II">World War II</span>,
 <span class="mw-headline" id="UNIVAC">UNIVAC</span>,
 <span class="mw-headline" id="COBOL">COBOL</span>,
 <span class="mw-headline" id="Standards">Standards</span>,
 <span class="mw-headline" id="Retirement">Retirement</span>,
 <span class="mw-headline" id="Post-retirement">Post-retirement</span>,
 <span class="mw-headline" id="Anecdotes">Anecdotes</span>,
 <span class="mw-headline" id="Death">Death</span>,
```

(continues on next page)

(continued from previous page)

```
 <span class="mw-headline" id="Dates_of_rank">Dates of rank</span>,
 <span class="mw-headline" id="Awards_and_honors">Awards and honors</span>,
 <span class="mw-headline" id="Military_awards">Military awards</span>,
 <span class="mw-headline" id="Other_awards">Other awards</span>,
 <span class="mw-headline" id="Legacy">Legacy</span>,
 <span class="mw-headline" id="Places">Places</span>,
 <span class="mw-headline" id="Programs">Programs</span>,
 <span class="mw-headline" id="In_popular_culture">In popular culture</span>,
 <span class="mw-headline" id="Grace_Hopper_Celebration_of_Women_in_Computing">Grace␣
↪Hopper Celebration of Women in Computing</span>,
 <span class="mw-headline" id="See_also">See also</span>,
 <span class="mw-headline" id="Notes">Notes</span>,
 <span class="mw-headline" id="References">References</span>,
 <span class="mw-headline" id="Obituary_notices">Obituary notices</span>,
 <span class="mw-headline" id="Further_reading">Further reading</span>,
 <span class="mw-headline" id="External_links">External links</span>]
```

[18]:
```
for item in soup.select(".mw-headline"):
    print(item.text)
```

```
Early life and education
Career
World War II
UNIVAC
COBOL
Standards
Retirement
Post-retirement
Anecdotes
Death
Dates of rank
Awards and honors
Military awards
Other awards
Legacy
Places
Programs
In popular culture
Grace Hopper Celebration of Women in Computing
See also
Notes
References
Obituary notices
Further reading
External links
```

Example Task 3 - Getting an Image from a Website

Let's attempt to grab the image of the Deep Blue Computer from this Wikipedia article: https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)

[19]:
```
res = requests.get("https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)")
```

[20]:
```
soup = bs4.BeautifulSoup(res.text,'lxml')
```

[23]:
```
# Find all tags with 'img' sub-string
img_tags = soup.find_all('img')
```

[24]:
```
# Find the tag with 'Deep_Blue' sub-string, which is the name of the computer.
for img in img_tags:
    # print(img['src'])
    if 'Deep_Blue' in img['src']:
        print(img['src'])
```

```
//upload.wikimedia.org/wikipedia/commons/thumb/b/be/Deep_Blue.jpg/220px-Deep_Blue.jpg
//upload.wikimedia.org/wikipedia/commons/thumb/8/83/One_of_Deep_Blue%27s_processors_
↪%282586060990%29.jpg/220px-One_of_Deep_Blue%27s_processors_%282586060990%29.jpg
```

We can actually display it with a markdown cell with the following:

```
<img src='https://upload.wikimedia.org/wikipedia/commons/thumb/b/be/Deep_Blue.jpg/220px-
↪Deep_Blue.jpg'>
```

Now that we have the actual link, we can grab the image with `requests` and `get`, along with the `.content` attribute. Note how we had to add `https://` before the link, because if you don't do this, requests will complain (but it gives you a pretty descriptive error code).

[25]:
```
image_link = requests.get('https://upload.wikimedia.org/wikipedia/commons/thumb/b/be/
↪Deep_Blue.jpg/220px-Deep_Blue.jpg')
```

[ ]:
```
# The raw content (it's a binary file, meaning we will need to use binary read/write
↪methods for saving it)
image_link.content
```

Let's write the image to a `jpg` file. Note the `'wb'` call to denote a binary writing of the file.

[27]:
```
f = open('my_new_file_name.jpg','wb')
```

[28]:
```
f.write(image_link.content)
```

[28]:
```
16806
```

[29]:
```
f.close()
```

Now we can display this file right here in the notebook as markdown using:

```
<img src="image/my_new_file_name.jpg" width="300">
```

Just write the above line in a new markdown cell and it will display the image we just downloaded.

Example Project - Working with Multiple Pages and Items

Let's show a more realistic example of scraping a full site. The website: http://books.toscrape.com/index.html is specifically designed for people to scrape it. Let's try to get the title of every book that has a 2 star rating and return a Python list with all their titles.

We will do the following:

1. Figure out the URL structure to go through every page
2. Scrape every page in the catalog
3. Figure out what tag/class represents the Star rating
4. Filter by that star rating using an if statement
5. Store the results in a list

We can see that the URL structure is the following:

```
http://books.toscrape.com/catalogue/page-1.html
```

[30]:
```
base_url = 'http://books.toscrape.com/catalogue/page-{}.html'
```

We can then fill in the page number with `.format()`.

[31]:
```
res = requests.get(base_url.format('1'))
```

Now let's grab the products (books) from the `get` request result.

[32]:
```
soup = bs4.BeautifulSoup(res.text,"lxml")
```

[ ]:
```
soup.select(".product_pod")
```

Now we can see that each book has the `product_pod` class. We can select any tag with this class, and then further reduce it by its rating.

[33]:
```
products = soup.select(".product_pod")
```

[34]:
```
example = products[0]
```

[35]:
```
type(example)
```

[35]:
```
bs4.element.Tag
```

[36]:
```
example.attrs
```

[36]:
```
{'class': ['product_pod']}
```

By inspecting the webpage we can see that the class we want is `class='star-rating Two'`. If you click on this in your browser, you'll notice it displays the space as a dot `.` , so that means we want to search for `".star-rating.Two"`.

[37]:
```
list(example.children)
```

[37]:
```
['\n',
 <div class="image_container">
 <a href="a-light-in-the-attic_1000/index.html"><img alt="A Light in the Attic" class=
↪"thumbnail" src="../media/cache/2c/da/2cdad67c44b002e7ead0cc35693c0e8b.jpg"/></a>
```

(continues on next page)

(continued from previous page)

```
</div>,
'\n',
<p class="star-rating Three">
<i class="icon-star"></i>
<i class="icon-star"></i>
<i class="icon-star"></i>
<i class="icon-star"></i>
<i class="icon-star"></i>
</p>,
'\n',
<h3><a href="a-light-in-the-attic_1000/index.html" title="A Light in the Attic">A Light␣
↪in the ...</a></h3>,
'\n',
<div class="product_price">
<p class="price_color">^Af51.77</p>
<p class="instock availability">
<i class="icon-ok"></i>

        In stock

</p>
<form>
<button class="btn btn-primary btn-block" data-loading-text="Adding..." type="submit">
↪Add to basket</button>
</form>
</div>,
'\n']
```

[38]:
```
example.select('.star-rating.Three')
```

[38]:
```
[<p class="star-rating Three">
 <i class="icon-star"></i>
 <i class="icon-star"></i>
 <i class="icon-star"></i>
 <i class="icon-star"></i>
 <i class="icon-star"></i>
 </p>]
```

But we are looking for 2 stars, so it looks like we can just check to see if something was returned.

[39]:
```
example.select('.star-rating.Two')
```

[39]:
```
[]
```

Alternatively, we can just quickly check the text string to see if `"star-rating Two"` is in it. Either approach is fine (there are also many other alternative approaches).

Now let's see how we can get the title if we have a 2-star match:

[40]:
```
example.select('a')
```

[40]:
```
[<a href="a-light-in-the-attic_1000/index.html"><img alt="A Light in the Attic" class=
↪"thumbnail" src="../media/cache/2c/da/2cdad67c44b002e7ead0cc35693c0e8b.jpg"/></a>,
 <a href="a-light-in-the-attic_1000/index.html" title="A Light in the Attic">A Light in␣
↪the ...</a>]
```

(continues on next page)

(continued from previous page)

[41]:
```
example.select('a')[1]
```

[41]:
```
<a href="a-light-in-the-attic_1000/index.html" title="A Light in the Attic">A Light in␣
↪the ...</a>
```

[42]:
```
example.select('a')[1]['title']
```

[42]:
```
'A Light in the Attic'
```

Be aware that a firewall may prevent this script from running. Also if you are getting a no response error, maybe try adding a sleep step with `time.sleep(1)`.

[43]:
```
two_star_titles = []

for n in range(1,51):

    scrape_url = base_url.format(n)
    res = requests.get(scrape_url)

    soup = bs4.BeautifulSoup(res.text,"lxml")
    books = soup.select(".product_pod")

    for book in books:
        if len(book.select('.star-rating.Two')) != 0:
            two_star_titles.append(book.select('a')[1]['title'])
```

[44]:
```
two_star_titles
```

[44]:
```
['Starving Hearts (Triangular Trade Trilogy, #1)',
 'Libertarianism for Beginners',
 "It's Only the Himalayas",
 'How Music Works',
 'Maude (1883-1993):She Grew Up with the country',
 "You can't bury them all: Poems",
 'Reasons to Stay Alive',
 'Without Borders (Wanderlove #1)',
 'Soul Reader',
 'Security',
 'Saga, Volume 5 (Saga (Collected Editions) #5)',
 'Reskilling America: Learning to Labor in the Twenty-First Century',
 'Political Suicide: Missteps, Peccadilloes, Bad Calls, Backroom Hijinx, Sordid Pasts,␣
↪Rotten Breaks, and Just Plain Dumb Mistakes in the Annals of American Politics',
 'Obsidian (Lux #1)',
 'My Paris Kitchen: Recipes and Stories',
 'Masks and Shadows',
 'Lumberjanes, Vol. 2: Friendship to the Max (Lumberjanes #5-8)',
 'Lumberjanes Vol. 3: A Terrible Plan (Lumberjanes #9-12)',
 'Judo: Seven Steps to Black Belt (an Introductory Guide for Beginners)',
 'I Hate Fairyland, Vol. 1: Madly Ever After (I Hate Fairyland (Compilations) #1-5)',
 'Giant Days, Vol. 2 (Giant Days #5-8)',
 'Everydata: The Misinformation Hidden in the Little Data You Consume Every Day',
```

(continues on next page)

(continued from previous page)

```
"Don't Be a Jerk: And Other Practical Advice from Dogen, Japan's Greatest Zen Master",
'Bossypants',
'Bitch Planet, Vol. 1: Extraordinary Machine (Bitch Planet (Collected Editions))',
'Avatar: The Last Airbender: Smoke and Shadow, Part 3 (Smoke and Shadow #3)',
'Tuesday Nights in 1980',
'The Psychopath Test: A Journey Through the Madness Industry',
'The Power of Now: A Guide to Spiritual Enlightenment',
"The Omnivore's Dilemma: A Natural History of Four Meals",
'The Love and Lemons Cookbook: An Apple-to-Zucchini Celebration of Impromptu Cooking',
'The Girl on the Train',
'The Emerald Mystery',
'The Argonauts',
'Suddenly in Love (Lake Haven #1)',
'Soft Apocalypse',
"So You've Been Publicly Shamed",
'Shoe Dog: A Memoir by the Creator of NIKE',
'Louisa: The Extraordinary Life of Mrs. Adams',
'Large Print Heart of the Pride',
'Grumbles',
'Chasing Heaven: What Dying Taught Me About Living',
'Becoming Wise: An Inquiry into the Mystery and Art of Living',
'Beauty Restored (Riley Family Legacy Novellas #3)',
'Batman: The Long Halloween (Batman)',
"Ayumi's Violin",
'Wild Swans',
"What's It Like in Space?: Stories from Astronauts Who've Been There",
'Until Friday Night (The Field Party #1)',
'Unbroken: A World War II Story of Survival, Resilience, and Redemption',
'Twenty Yawns',
'Through the Woods',
'This Is Where It Ends',
'The Year of Magical Thinking',
'The Last Mile (Amos Decker #2)',
'The Immortal Life of Henrietta Lacks',
'The Hidden Oracle (The Trials of Apollo #1)',
'The Guilty (Will Robie #4)',
'Red Hood/Arsenal, Vol. 1: Open for Business (Red Hood/Arsenal #1)',
'Once Was a Time',
'No Dream Is Too High: Life Lessons From a Man Who Walked on the Moon',
'Naruto (3-in-1 Edition), Vol. 14: Includes Vols. 40, 41 & 42 (Naruto: Omnibus #14)',
'More Than Music (Chasing the Dream #1)',
'Lowriders to the Center of the Earth (Lowriders in Space #2)',
'Eat Fat, Get Thin',
'Doctor Sleep (The Shining #2)',
'Crazy Love: Overwhelmed by a Relentless God',
'Carrie',
'Batman: Europa',
'Angels Walking (Angels Walking #1)',
'Adulthood Is a Myth: A "Sarah\'s Scribbles" Collection',
'A Study in Scarlet (Sherlock Holmes #1)',
'A Series of Catastrophes and Miracles: A True Story of Love, Science, and Cancer',
"A People's History of the United States",
```

(continues on next page)

(continued from previous page)

```
 'My Kitchen Year: 136 Recipes That Saved My Life',
 'The Lonely City: Adventures in the Art of Being Alone',
 'The Dinner Party',
 'Stars Above (The Lunar Chronicles #4.5)',
 'Love, Lies and Spies',
 'Troublemaker: Surviving Hollywood and Scientology',
 'The Widow',
 'Setting the World on Fire: The Brief, Astonishing Life of St. Catherine of Siena',
 'Mothering Sunday',
 'Lilac Girls',
 '10% Happier: How I Tamed the Voice in My Head, Reduced Stress Without Losing My Edge,␣
↪and Found Self-Help That Actually Works',
 'Underlying Notes',
 'The Flowers Lied',
 'Modern Day Fables',
 "Chernobyl 01:23:40: The Incredible True Story of the World's Worst Nuclear Disaster",
 '23 Degrees South: A Tropical Tale of Changing Whether...',
 'When Breath Becomes Air',
 'Vagabonding: An Uncommon Guide to the Art of Long-Term World Travel',
 'The Martian (The Martian #1)',
 "Miller's Valley",
 "Love That Boy: What Two Presidents, Eight Road Trips, and My Son Taught Me About a␣
↪Parent's Expectations",
 'Left Behind (Left Behind #1)',
 'Howl and Other Poems',
 "Heaven is for Real: A Little Boy's Astounding Story of His Trip to Heaven and Back",
 "Brazen: The Courage to Find the You That's Been Hiding",
 '32 Yolks',
 'Wildlife of New York: A Five-Borough Coloring Book',
 'Unreasonable Hope: Finding Faith in the God Who Brings Purpose to Your Pain',
 'The Art Book',
 'Steal Like an Artist: 10 Things Nobody Told You About Being Creative',
 'Raymie Nightingale',
 'Like Never Before (Walker Family #2)',
 'How to Be a Domestic Goddess: Baking and the Art of Comfort Cooking',
 'Finding God in the Ruins: How God Redeems Pain',
 'Chronicles, Vol. 1',
 'A Summer In Europe',
 'The Rise and Fall of the Third Reich: A History of Nazi Germany',
 'The Makings of a Fatherless Child',
 'The Fellowship of the Ring (The Lord of the Rings #1)',
 "Tell the Wolves I'm Home",
 'In the Woods (Dublin Murder Squad #1)',
 'Give It Back',
 'Why Save the Bankers?: And Other Essays on Our Economic and Political Crisis',
 'The Raven King (The Raven Cycle #4)',
 'The Expatriates',
 'The 5th Wave (The 5th Wave #1)',
 'Peak: Secrets from the New Science of Expertise',
 'Logan Kade (Fallen Crest High #5.5)',
 "I Know Why the Caged Bird Sings (Maya Angelou's Autobiography #1)",
 'Drama',
```

(continues on next page)

(continued from previous page)

```
"America's War for the Greater Middle East: A Military History",
'A Game of Thrones (A Song of Ice and Fire #1)',
"The Pilgrim's Progress",
'The Hound of the Baskervilles (Sherlock Holmes #5)',
"The Geography of Bliss: One Grump's Search for the Happiest Places in the World",
'The Demonists (Demonist #1)',
'The Demon Prince of Momochi House, Vol. 4 (The Demon Prince of Momochi House #4)',
'Misery',
'Far From True (Promise Falls Trilogy #2)',
'Confessions of a Shopaholic (Shopaholic #1)',
'Vegan Vegetarian Omnivore: Dinner for Everyone at the Table',
'Two Boys Kissing',
'Twilight (Twilight #1)',
'Twenties Girl',
'The Tipping Point: How Little Things Can Make a Big Difference',
'The Stand',
'The Picture of Dorian Gray',
'The Name of God is Mercy',
"The Lover's Dictionary",
'The Last Painting of Sara de Vos',
'The Guns of August',
'The Girl Who Played with Fire (Millennium Trilogy #2)',
'The Da Vinci Code (Robert Langdon #2)',
'The Cat in the Hat (Beginner Books B-1)',
'The Book Thief',
'The Autobiography of Malcolm X',
"Surely You're Joking, Mr. Feynman!: Adventures of a Curious Character",
'Soldier (Talon #3)',
'Shopaholic & Baby (Shopaholic #5)',
'Seven Days in the Art World',
'Rework',
'Packing for Mars: The Curious Science of Life in the Void',
'Orange Is the New Black',
'One for the Money (Stephanie Plum #1)',
'Midnight Riot (Peter Grant/ Rivers of London - books #1)',
'Me Talk Pretty One Day',
'Manuscript Found in Accra',
'Lust & Wonder',
"Life, the Universe and Everything (Hitchhiker's Guide to the Galaxy #3)",
'Life After Life',
'I Am Malala: The Girl Who Stood Up for Education and Was Shot by the Taliban',
'House of Lost Worlds: Dinosaurs, Dynasties, and the Story of Life on Earth',
'Horrible Bear!',
'Holidays on Ice',
'Girl in the Blue Coat',
'Fruits Basket, Vol. 3 (Fruits Basket #3)',
'Cosmos',
'Civilization and Its Discontents',
"Catastrophic Happiness: Finding Joy in Childhood's Messy Years",
'Career of Evil (Cormoran Strike #3)',
'Born to Run: A Hidden Tribe, Superathletes, and the Greatest Race the World Has Never␣
↪Seen',
```

(continues on next page)

(continued from previous page)

```
"Best of My Love (Fool's Gold #20)",
'Beowulf',
'Awkward',
'And Then There Were None',
'A Storm of Swords (A Song of Ice and Fire #3)',
'The Suffragettes (Little Black Classics, #96)',
'Vampire Girl (Vampire Girl #1)',
'Three Wishes (River of Time: California #1)',
'The Wicked + The Divine, Vol. 1: The Faust Act (The Wicked + The Divine)',
'The Little Prince',
'The Last Girl (The Dominion Trilogy #1)',
'Taking Shots (Assassins #1)',
'Settling the Score (The Summer Games #1)',
'Rhythm, Chord & Malykhin',
'One Second (Seven #7)',
"Old Records Never Die: One Man's Quest for His Vinyl and His Past",
'Of Mice and Men',
'My Perfect Mistake (Over the Top #1)',
'Meditations',
'Frankenstein',
'Emma']
```

Now you should have the tools necessary to scrape any websites that interest you.

Keep in mind that the more complex the website, the harder it will be to scrape.

And always ask for permission!

BACK TO TOP

7.33 Tutorial 6 - Google Colab

Google Colab (or Colaboratary) is a cloud-based platform created by Google that offers an environment for sharing, running, and writing Python code within Google Drive. Colab runs Jupyter notebook files, and it comes with pre-installed popular Data Science and Machine Learning libraries and frameworks, such as TensorFlow, PyTorch, NumPy, pandas, and others.

Google Colab provides CPU, TPU, and GPU support. It enables real-time collaborative editing on a single notebook, much like the collaborative text editing functionality provided by Google Docs.

This is the official Colab webpage.

Welcome Page of Colab

The welcome page offers basic tutorials about working with Jupyter notebooks, data science, and machine learning.

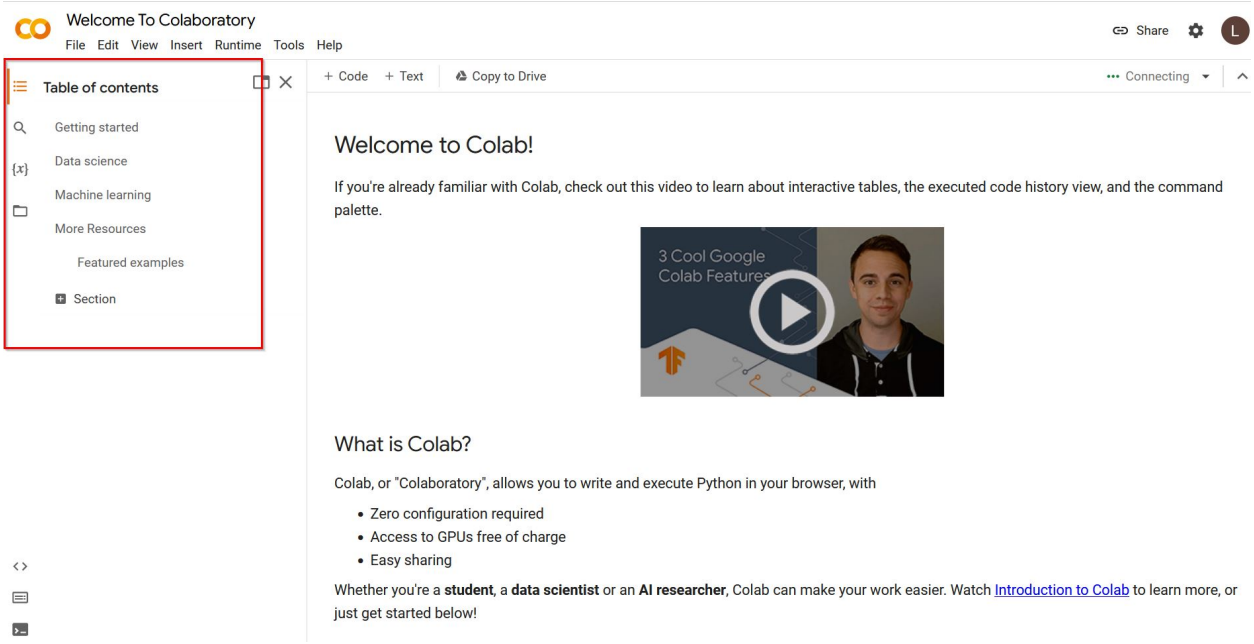
It provides:

- Overview of Colab basic features, such as Code Cells and Markdown Cells
- Loading data: Drive, Sheets, and Google Cloud Storage Link
- Data Visualization Link
- Machine Learning introduction course Link

Top Menus

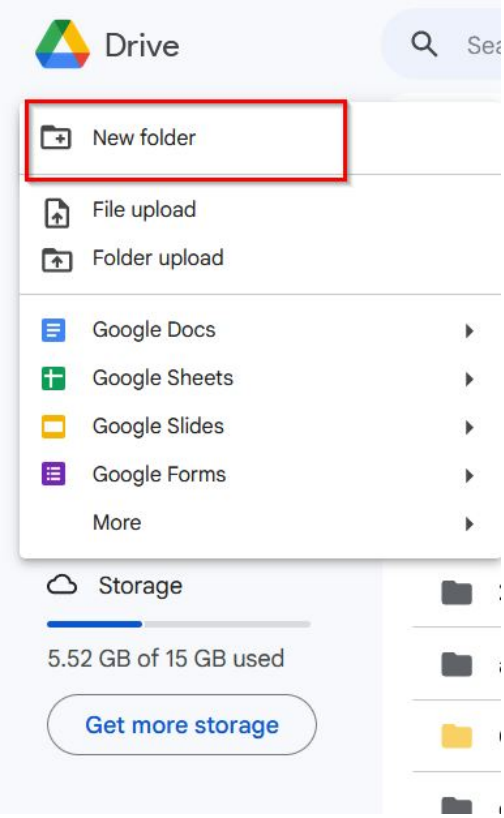
On the top menu section, Colab provides very similar features to the original Jupyter Notebook interface.

- **File**: create/rename/upload/move/save notebook files
- **Edit**: move/copy/past cells, notebook settings
- **Insert**: code/text/section header cells/code snippets/add a form field
- **Runtime**: run/interrupt/restart cells/runtime type change (GPU<->CPU)
- **Tools**: command palette/settings/keyboard shortcuts, etc.

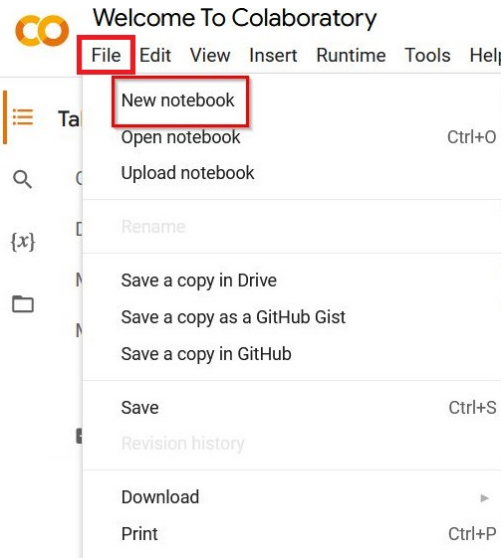


7.33.1 Upload Files and Mount the Google Drive

1) Login into Google Drive with your Gmail account (Link).
2) Create a folder to store your data files.



3) Create a new Jupyter notebook file. The file can be accessed in the "Colab Notebooks" folder in Google Drive.

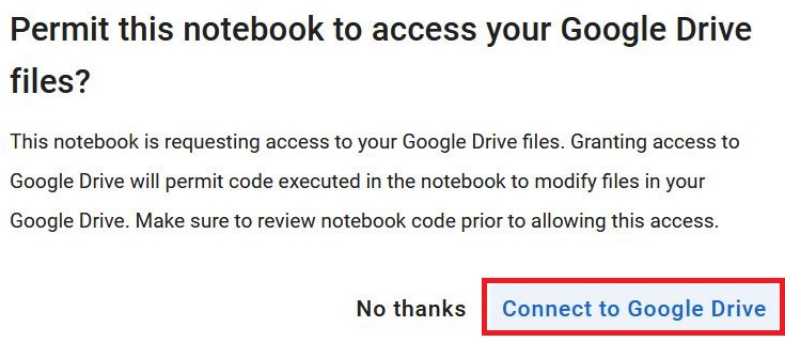


4) Use the following code to mount your Google Drive in the Jupyter notebook.

```
[6]: from google.colab import drive
     drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

5) Click `Connect to Google Drive` to permit the notebook to access Google Drive.



6) After you mount the Google Drive, you can load files from the drive. For example, load a csv file. Note that the path to the file needs to start with `drive/My Drive/...`.

[7]:
```
import pandas as pd
df_IMDb = pd.read_csv('drive/My Drive/data_file/IMDb_movies.csv')
df_IMDb
```

[7]:

	Unnamed: 0	Movie Name	Year of Release	Watch Time	Movie Rating	Metascore of movie	Gross	Votes	Description
0	0	The Shawshank Redemption	1994	142	9.3	82.0	28.34	27,77,378	Over the course of several years, two convicts...
1	1	The Godfather	1972	175	9.2	100.0	134.97	19,33,588	Don Vito Corleone, head of a mafia family, dec...
2	2	The Dark Knight	2008	152	9.0	84.0	534.86	27,54,087	When the menace known as the Joker wreaks havo...
3	3	Schindler's List	1993	195	9.0	95.0	96.9	13,97,886	In German-occupied Poland during World War II,...
4	4	12 Angry Men	1957	96	9.0	97.0	4.36	8,24,211	The jury in a New York City murder trial is fr...
..	...	...	...	...	...	...	...	...	...
995	995	Philomena	2013	98	7.6	77.0	37.71	1,02,336	A world-weary political journalist picks up th...
996	996	Un long dimanche de fiançailles	2004	133	7.6	76.0	6.17	75,004	Tells the story of a young woman's relentless ...
997	997	Shine	1996	105	7.6	87.0	35.81	55,589	
998	998	The Invisible Man	1933	71	7.6	87.0	NaN	37,822	
999	999	Celda 211	2009	113	7.6	NaN	NaN	69,464	

(continues on next page)

(continued from previous page)

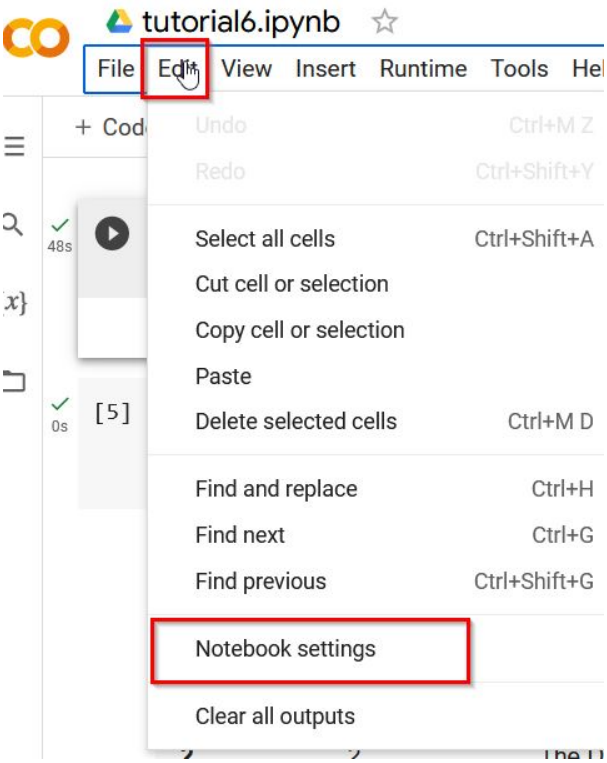
```
997  Pianist David Helfgott, driven by his father a...
998  A scientist finds a way of becoming invisible,...
999  The story of two men on different sides of a p...

[1000 rows x 9 columns]
```

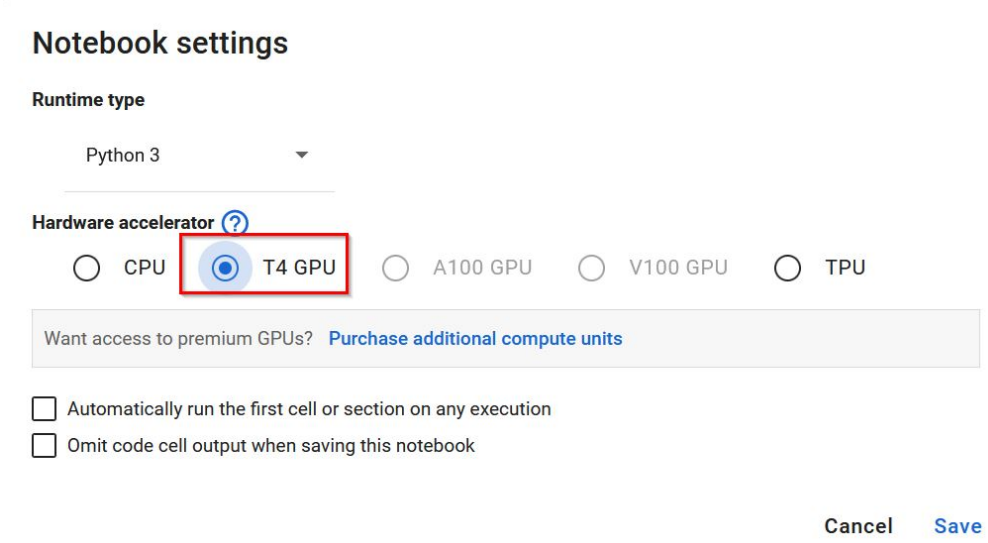
Enable GPU

Colab has built-in features that allow users to switch between CPU and GPU for working with Data Science/Machine Learning models.

Method1:

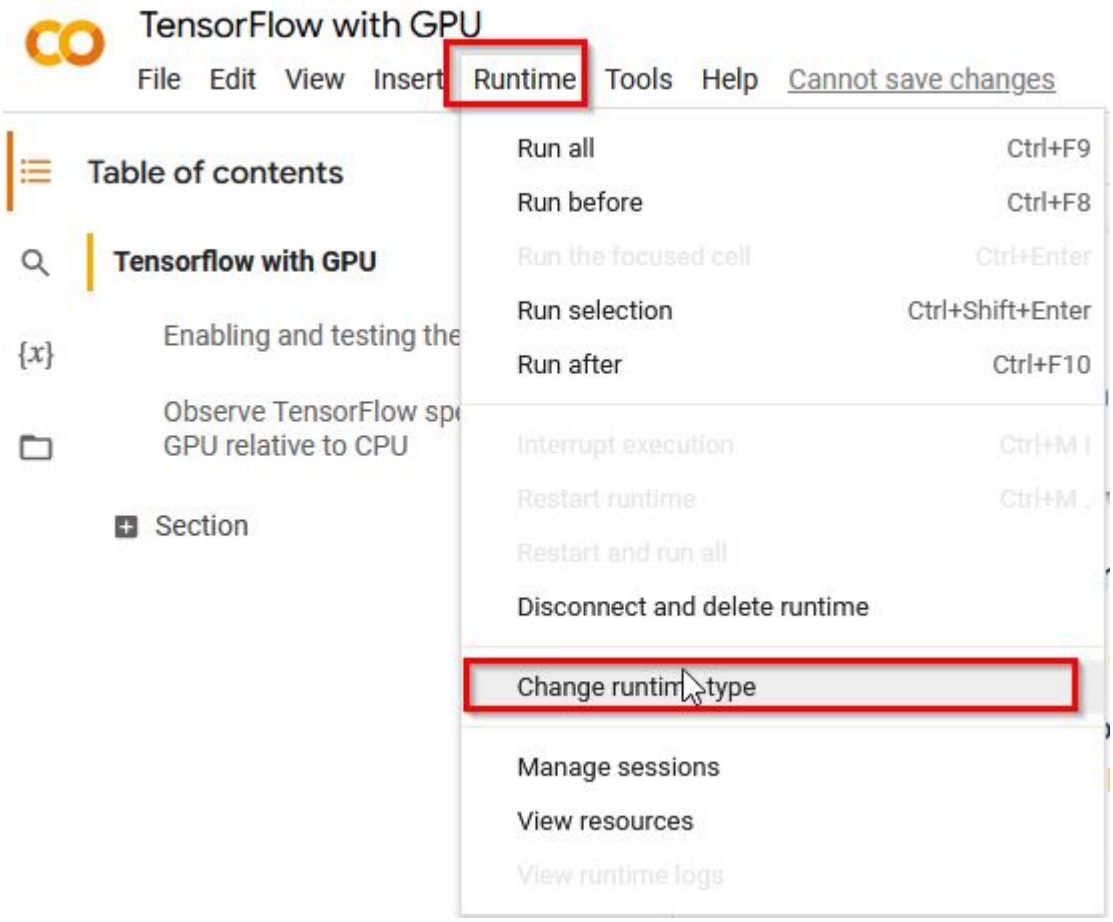
- Click `Edit` → `Notebook Settings`.
- Choose an available GPU from the `Hardware Accelerator`, and click `Save`.





Method2:

- Click `Runtime` → `Change runtime type`.
- Choose an available GPU from the `Hardware Accelerator`, and click `Save`.



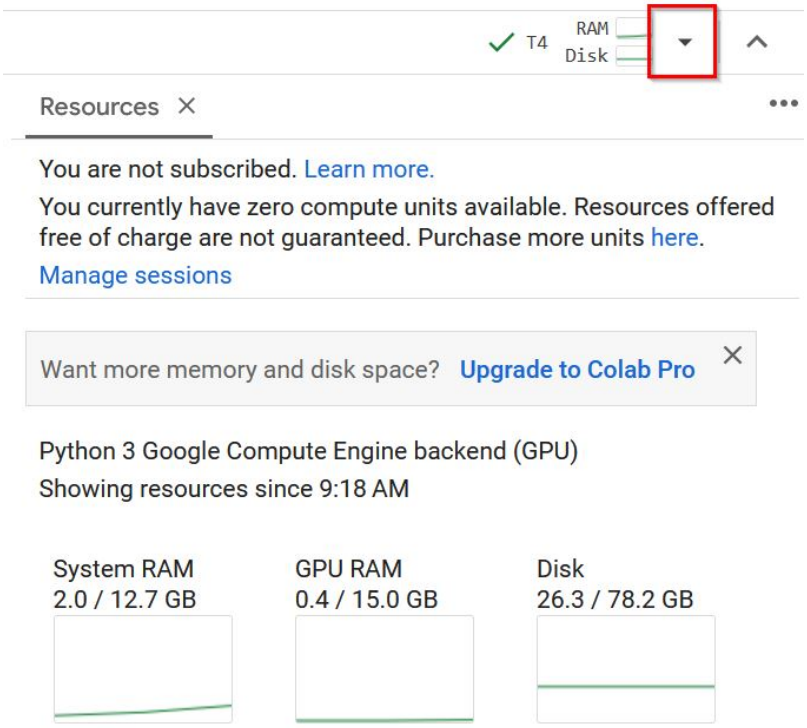
Using GPU

Use the following code to load the GPU in your script. When you are connected to GPU, the code will print out "Found GPU at: /device:GPU:0"

[4]:
```
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

Monitor Your Hardware Resources

To monitor available and used resources, click the `Connect` button in the upper right corner of your screen, and if your GPU is connected, it will show the hardware stats.

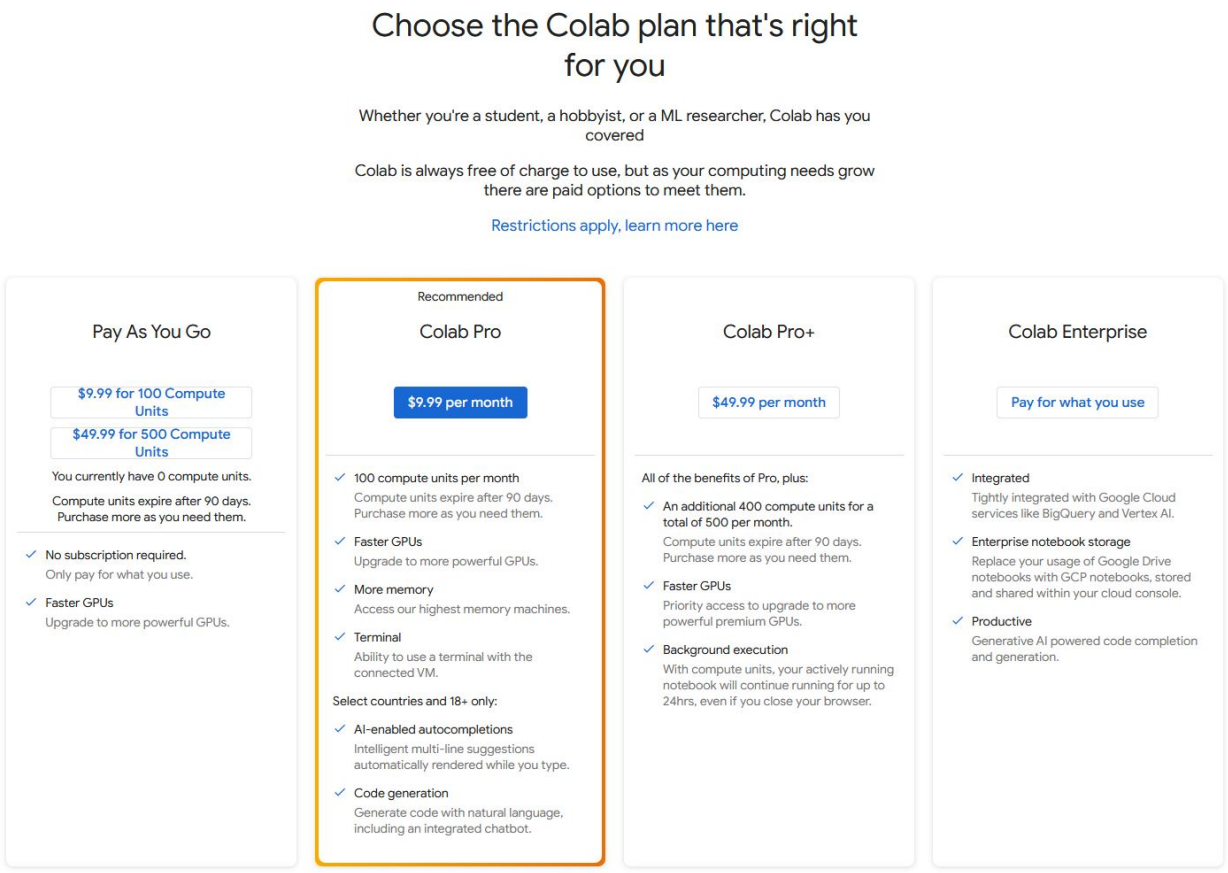


Colab Subscription Plans

Paid Colab plans provide better hardware, i.e. access to more powerful GPU, and more VRAM and RAM. They also have longer timeout sessions. For instance, Colab Pro+ allows you to run the script for up to 24 hours with the web browser closed.

The free Colab version offers 16GB of GPU RAM, while the paid versions can have up to 48GB of RAM. Larger VRAM and RAM may be required to train some large language models (LLM).

For the free version, only T4 GPU are available, but sometimes there would be no GPU available. The free version has at most 12 hours sessions before timeout. In practice, for the free version, your session could be timed out and your script could be interrupted at any time. If you reconnect within the next 1-2 minutes, the session can be resumed, and otherwise, you will have to re-run your script.



BACK TO TOP

7.34 Tutorial 7- Image Processing with Python

A **digital image** is a digital or electronic representation of a visual scene or object, typically constructed from a grid of individual picture elements, referred to as *pixels*. Each pixel represents a tiny square or point, that stores data related to the color and luminance of a particular location within the image. Digital images serve diverse purposes across many applications, encompassing photography, graphic design, video games, medical imaging, and other fields.

Digital image processing is the computational approach for the analysis, storage, and interpretation of digital content. In modern times, the impact of image processing has expanded to include many fields ranging from medical diagnostics to autonomous navigation.

Popular libraries for digital image processing include Pillow, OpenCV, and Scikit-image.

7.34.1 Python Image Processing with Pillow

Pillow is a popular library for performing advanced image processing tasks that do not require extensive expertise in the field. It is also commonly employed for initial experimentation with image-related tasks. Additionally, Pillow benefits from its widespread adoption within the Python community and offers a more gentle learning curve compared to some of the more complex image processing libraries.

To install Pillow in Python use:

```
pip install pillow
```

Load and Display an Image with Pillow

The methods `open()` and `load()` are used for loading images in Pillow.

The `display()` method stores the image as a temporary file and then displays it through the native software of the operating system designed for handling images.

[1]:
```
from PIL import Image

img_name = "images/UI.png"

# open and load the image object
with Image.open(img_name) as img:
    img.load()

# display the image
display(img)
```

We can print the type of an image object as in the following cell.

[2]:
```
type(img)
```

[2]:
```
PIL.PngImagePlugin.PngImageFile
```

Save an image

The `save()` method saves an image by specifying a string with a name for the image file, which can also include the path for saving the image.

[3]:
```
img.save("saved_img.png")
```

The following code prints the size and format of the image.

[4]:
```
# print the format of the image
img.format
```

[4]:
```
'PNG'
```

[5]:
```
# print the width and height of the image in pixels
img.size
```

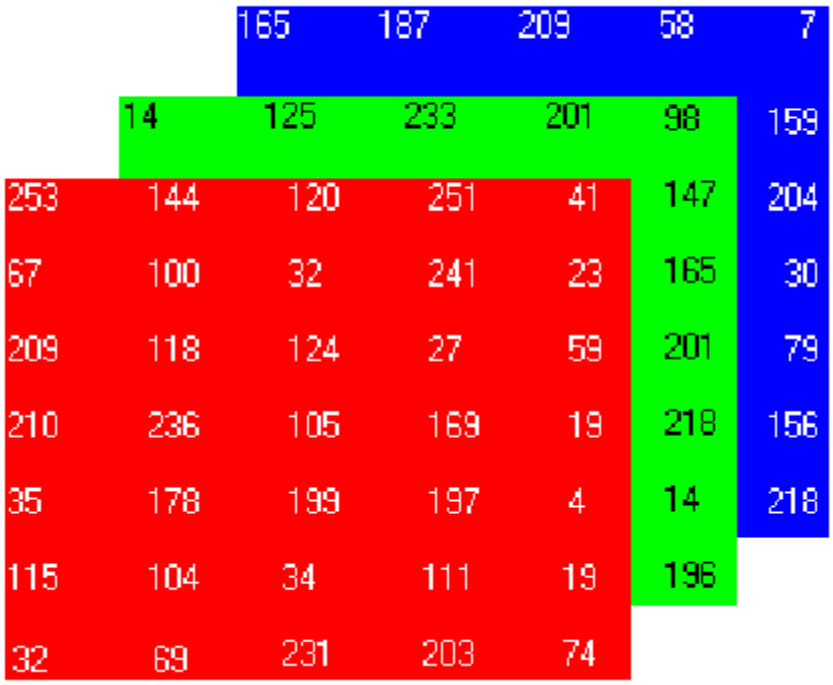
[5]:
```
(1019, 569)
```

Channels and Mode of an Image

An image is a 2D grid of pixels, with each pixel denoting a specific color. The pixels can be defined using one or more values. In the case of an RGB image, each pixel is described by three values, representing its red, green, and blue components. Consequently, an Image object for an RGB image comprises three `channels` or `bands`, each corresponding to a color component.

For instance, in a 200x200-pixel RGB image, the representation is an array of 200 x 200 x 3 values.

An example of an RGB array is shown in the figure.



[6]:
```
# print the channels of a RGB image
img.getbands()
```

[6]:
```
('R', 'G', 'B')
```

The `mode` specifies the type of an image. Pillow offers a broad range of standard modes, such as black-and-white (binary), grayscale, RGB, RGBA, and CMYK.

[7]:
```
# print the mode of the image
img.mode
```

[7]:
```
'RGB'
```

To convert the RGB image into grayscale mode, we can use `convert("L")`.

[8]:
```
# Convert to grayscale image
gray_img = img.convert("L")
display(gray_img)
```

Show the bands of a grayscale mode image.

[9]:
```
gray_img.getbands()
```

[9]:
```
('L',)
```

Split and Merge Channels

You can separate an image into its channels using `split()` and we can combine the separate channels back into an Image object using `merge()`.

The `split()` method returns all the channels as separate Image objects. We can confirm this by displaying the mode of one of the returned objects.

[10]:
```
# Split the channels
red, green, blue = img.split()
red.mode
```

[10]:
```
'L'
```

The first argument in `merge()` determines the mode of the image that we want to create. The second argument contains the individual channels that we want to merge into a single image.

In the next cell, the red channel is stored in the variable `red`, and it is a grayscale image with mode `'L'`. To create an image showing only the red channel, we merge the red channel from the original image with green and blue channels that contain only zeros. And, to create a channel containing zeros everywhere, we use the `point()` method.

[11]:
```
zeroed_band = red.point(lambda _: 0)
print('Show the red channel:')
red_merge = Image.merge("RGB", (red, zeroed_band, zeroed_band))
display(red_merge)
```

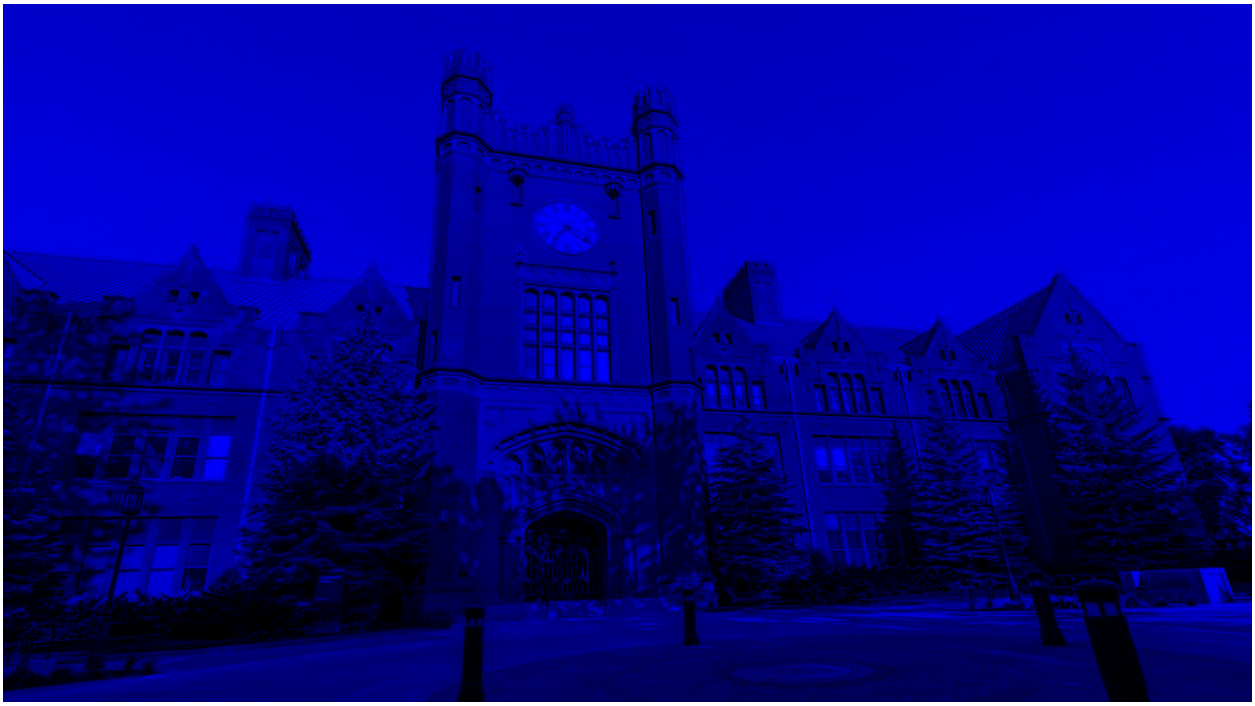
Show the red channel:

[12]:
```
print('Show the green channel')
green_merge = Image.merge("RGB", (zeroed_band, green, zeroed_band))
display(green_merge)
```

Show the green channel

[13]:
```
print('Show the blue channel:')
blue_merge = Image.merge("RGB", (zeroed_band, zeroed_band, blue))
display(blue_merge)
```

```
Show the blue channel:
```



Basic Image Manipulation

The `transpose()` method in Pillow allows to rotate or flip an image.

Options in the `transpose()` method include:

- `Image.FLIP_LEFT_RIGHT`: Flips the image left to right, resulting in a mirror image.
- `Image.FLIP_TOP_BOTTOM`: Flips the image top to bottom.
- `Image.ROTATE_90`: Rotates the image by 90 degrees counterclockwise.
- `Image.ROTATE_180`: Rotates the image by 180 degrees.
- `Image.ROTATE_270`: Rotates the image by 270 degrees counterclockwise, which is the same as 90 degrees clockwise.
- `Image.TRANSPOSE`: Transposes the rows and columns using the top-left pixel as the origin, with the top-left pixel being the same in the transposed image as in the original image.
- `Image.TRANSVERSE`: Transposes the rows and columns using the bottom-left pixel as the origin, with the bottom-left pixel being the one that remains fixed between the original and modified versions.

An example with `FLIP_TOP_BOTTOM`.

[14]:
```
converted_img = img.transpose(Image.FLIP_TOP_BOTTOM)
display(converted_img)
```

```
C:\Users\avaka\AppData\Local\Temp\ipykernel_16052\148912711.py:1: DeprecationWarning:␣
↪FLIP_TOP_BOTTOM is deprecated and will be removed in Pillow 10 (2023-07-01). Use␣
↪Transpose.FLIP_TOP_BOTTOM instead.
  converted_img = img.transpose(Image.FLIP_TOP_BOTTOM)
```

Example of rotating the image for 60 degrees counterclockwise.

[15]:
```
rotated_img = img.rotate(60)
display(rotated_img)
```

The Image object returned is the same size as the original Image. Therefore, the corners of the image are missing in the image displayed above.

We can change this behavior with the parameter `expand` set to True.

[16]:
```
rotated_img = img.rotate(45, expand=True)
display(rotated_img)
```

Crop and Resize Images

We can crop images using the `crop()` method. The arguments in `crop()` must comprise a 4 element tuple that defines the left, upper, right, and bottom edges of the region that we wish to crop. The coordinate system used in Pillow assigns the coordinates (0, 0) to the pixel in the upper-left corner.

[17]:
```
cropped_img = img.crop((400, 300, 600, 500))
display(cropped_img)
```

[18]:
```
cropped_img.size
```

[18]:
```
(200, 200)
```

To resize an image, we can change the resolution with the `resize()` method. For instance, we can set the new width and height to half of their original values using the floor division operator (`//`) and the Image attributes `.width` and `.height`.

[19]:
```
low_res_img = cropped_img.resize((cropped_img.width // 2, cropped_img.height // 2))
display(low_res_img)
```

Image Blurring, Sharpening, and Smoothing

We can blur an image by using the `filter()` method and the `ImageFilter` module.

[20]:
```
from PIL import ImageFilter
blur_img = img.filter(ImageFilter.BLUR)
display(blur_img)
```

More advanced blurring methods include:

- Gaussian Blur using `.GaussianBlur(factor)` method
- Box blur using `.BoxBlur(factor)` method

The argument in the blur methods determines how much blurring to apply. The larger the factor, the more blur is added to the image.

[21]:
```
print('BoxBlur method with factor of 5')
display(img.filter(ImageFilter.BoxBlur(5)))
```

```
BoxBlur method with factor of 5
```

[22]:
```
print('BoxBlur method with factor of 20')
display(img.filter(ImageFilter.BoxBlur(20)))
```

```
BoxBlur method with factor of 20
```

[23]:
```
print('GaussianBlur method with factor of 20')
display(img.filter(ImageFilter.GaussianBlur(20)))
```

```
GaussianBlur method with factor of 20
```

To sharpen an image, we can use the predefined filter `ImageFilter.SHARPEN`.

[24]:
```
# original image
display(cropped_img)
```

[25]:
```
# sharpened image
sharp_img = cropped_img.filter(ImageFilter.SHARPEN)
display(sharp_img)
```

Another more advanced sharpening method is with `ImageEnhance.Sharpness(factor)`. Applying sharpening with a factor greater than 1 applies a sharpening filter to the image, sharpening factor in the range from 0 to 1 blurs the image, and sharpening factor = 1 returns the original image.

[26]:
```
from PIL import ImageEnhance
sharp_image =ImageEnhance.Sharpness(cropped_img).enhance(4)
display(sharp_image)
```

Change Color, Brightness, Contrast

`ImageEnhance.Color(img).enhance(factor)` changes the color of the image, where factor > 1 means stronger color, factor < 1 reduces the color, and 0 means grayscale image.

`ImageEnhance.Color(im).Brightness(factor)` changes the brightness of the image, where factor > 1 makes the image brighter, factor < 1 makes it darker, and 0 means black image.

`ImageEnhance.Contrast(im).enhance(factor)` changes the contrast, where factor > 1 increases the brightness range, making light colors k-times lighter and dark colors darker. At very high contrast values, every pixel is either black or white, and generally only the basic shapes of the image are visible. Factor less than 1 decreases the brightness range, pulling all the colors towards a middle grey. A factor of 0 results in a completely grey image.

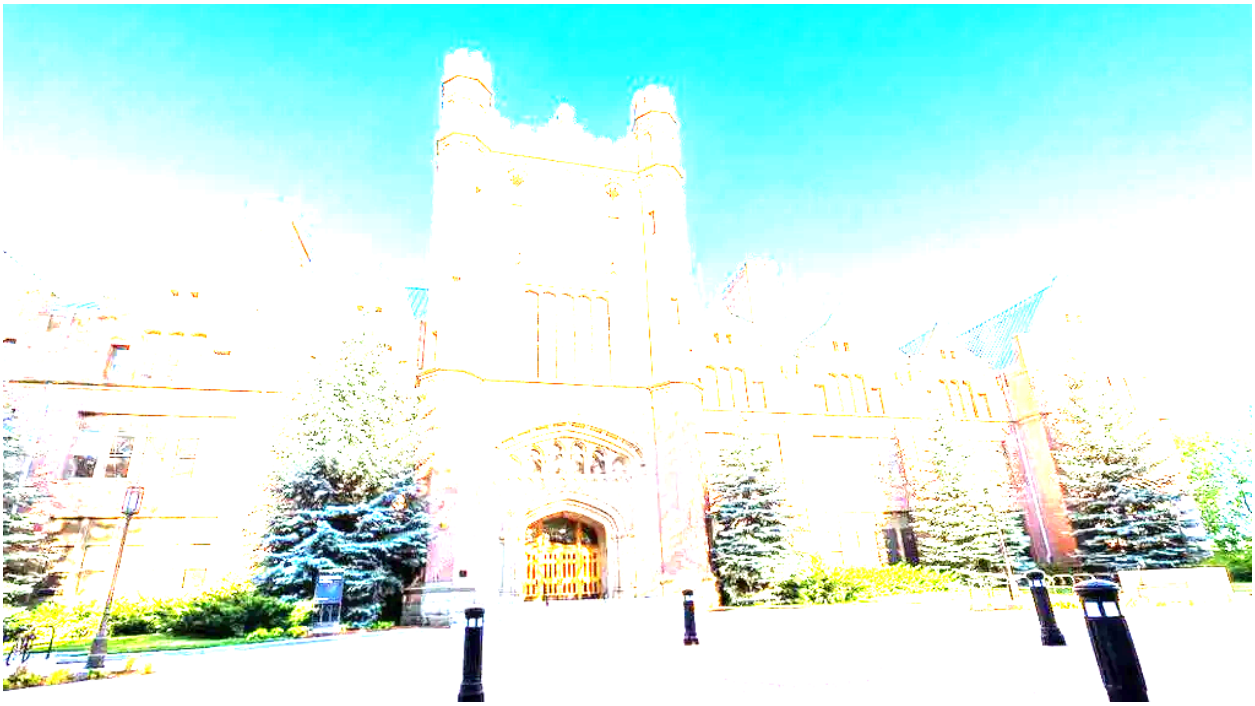
Change the color of an image.

[27]:
```
# Make the color stronger
color_img = ImageEnhance.Color(img).enhance(5)
display(color_img)
```

[28]:
```
# Make the color less strong
less_color_img = ImageEnhance.Color(img).enhance(0.5)
display(less_color_img)
```

Change the brightness of an image.

[29]:
```
# Make the image brighter
brighter_img = ImageEnhance.Brightness(img).enhance(5)
display(brighter_img)
```



[30]:
```
# Make the image darker
darker_img = ImageEnhance.Brightness(img).enhance(.5)
display(darker_img)
```

Change the contrast of an image.

[31]:
```
# Apply more contrast
contra_img = ImageEnhance.Contrast(img).enhance(5)
display(contra_img)
```

[32]:
```
# Apply less contrast
less_contra_img = ImageEnhance.Contrast(img).enhance(.5)
display(less_contra_img)
```

References:

1. "ELEC_ENG 420: Digital Image Processing | Electrical and Computer Engineering | Northwestern Engineering" available at www.mccormick.northwestern.edu/electrical-computer/academics/courses/descriptions/420.html.
2. "Image Processing With the Python Pillow Library," Real Python, available at https://realpython.com/image-processing-with-the-python-pillow-library/.
3. "Python Pillow Tutorial," GeeksforGeeks, available at https://www.geeksforgeeks.org/python-pillow-tutorial/.

BACK TO TOP

7.35 Tutorial 8 - TensorFlow

This tutorial is adapted from a 2022 blog post on the website Made With ML, by Goku Mohandas.

We will import `TensorFlow` and `numpy` and set the seed for their random number generators for reproducibility.

[2]:
```
import numpy as np
import tensorflow as tf

tf_version = tf.__version__

print(f'TF version: {tf_version}')
```

```
TF version: 2.8.2
```

[2]:
```
SEED = 1
```

[3]:
```
# Set seed for reproducability
np.random.seed(seed=SEED)
tf.random.set_seed(seed=SEED)
```

7.35.1 Basics

First, we will cover some basics such as creating TensorFlow tensors and converting from common data structures to TensorFlow tensors.

[4]:
```
x = tf.random.normal((2, 3))
print(f'Type: {x.dtype}')
print(f'shape: {x.shape}')
print(f'values:\n{x}')
```

```
Type: <dtype: 'float32'>
shape: (2, 3)
values:
[[-1.1012203   1.5457517   0.383644  ]
 [-0.87965786 -1.2246722  -0.9811211 ]]
```

[ ]:
```
# Zeros and ones tensors
x = tf.zeros((2, 3), dtype=tf.float64)
```

(continues on next page)

(continued from previous page)

```
print(x)
x = tf.ones((2, 3))
print(x)
```

[ ]:
```
# List => Tensor
x = tf.constant([[1, 2, 3],
                 [4, 5, 6]])

print(f'shape: {x.shape}')
print(f'values:\n{x}')
```

[8]:
```
# Numpy array => Tensor
x = tf.constant(np.random.rand(2, 3))
print(f'shape: {x.shape}')
print(f'values:\n{x}')
```

```
shape: (2, 3)
values:
[[4.17022005e-01 7.20324493e-01 1.14374817e-04]
 [3.02332573e-01 1.46755891e-01 9.23385948e-02]]
```

7.35.2 Operations

[9]:
```
# Addition
x = tf.random.normal((2, 3))
y = tf.random.normal((2, 3))
z = x + y
print(f'shape: {z.shape}')
print(f'values:\n{z}')
```

```
shape: (2, 3)
values:
[[-0.05392435 -1.4948881   0.6654824 ]
 [ 0.4435789   1.0243716   0.5050061 ]]
```

[10]:
```
# Dot product
x = tf.random.normal((2, 3))
y = tf.random.normal((3, 2))
z = tf.matmul(x, y)
print(f'shape: {z.shape}')
print(f'values:\n{z}')
```

```
shape: (2, 2)
values:
[[ 2.04325     1.1587756 ]
 [-1.8957058  -0.67201185]]
```

[11]:
```
# Transpose
x = tf.random.normal((2, 3))
print(f"shape: {x.shape}")
print(f"values: \n{x}")
```

(continues on next page)

(continued from previous page)

```
y = tf.transpose(x)
print(f"shape: {y.shape}")
print(f"values: \n{y}")
```

```
shape: (2, 3)
values:
[[-1.1771783  -0.90325946  0.8419609 ]
 [-0.06870949 -0.96161884 -0.51533026]]
shape: (3, 2)
values:
[[-1.1771783  -0.06870949]
 [-0.90325946 -0.96161884]
 [ 0.8419609  -0.51533026]]
```

[3]:
```
# Reshape
x = tf.random.normal((2, 3))
z = tf.reshape(x, (3, 2))
print(f"shape: {z.shape}")
print(f"values: \n{z}")
```

```
shape: (3, 2)
values:
[[ 0.73264295  0.99835527]
 [ 1.555784    1.0374023 ]
 [-0.0362004  -0.18817899]]
```

[14]:
```
# Dangers of reshaping (unintended consequences)
x = tf.constant([
    [[1,1,1,1], [2,2,2,2], [3,3,3,3]],
    [[10,10,10,10], [20,20,20,20], [30,30,30,30]]
])
print(f"shape: {x.shape}")
print(f"x: \n{x}\n")

a = tf.reshape(x, (x.shape[1], -1))
print(f"\nshape: {a.shape}")
print(f"a: \n{a}\n")

b = tf.transpose(x, perm=[1, 0, 2])
print(f"\nshape: {b.shape}")
print(f"b: \n{b}\n")

c = tf.reshape(b, (b.shape[0], -1))
print(f"\nshape: {c.shape}")
print(f"c: \n{c}")
```

```
shape: (2, 3, 4)
x:
[[[ 1  1  1  1]
  [ 2  2  2  2]
  [ 3  3  3  3]]

 [[10 10 10 10]
```

(continues on next page)

(continued from previous page)

```
  [20 20 20 20]
  [30 30 30 30]]]


shape: (3, 8)
a:
[[ 1  1  1  1  2  2  2  2]
 [ 3  3  3  3 10 10 10 10]
 [20 20 20 20 30 30 30 30]]


shape: (3, 2, 4)
b:
[[[ 1  1  1  1]
  [10 10 10 10]]

 [[ 2  2  2  2]
  [20 20 20 20]]

 [[ 3  3  3  3]
  [30 30 30 30]]]


shape: (3, 8)
c:
[[ 1  1  1  1 10 10 10 10]
 [ 2  2  2  2 20 20 20 20]
 [ 3  3  3  3 30 30 30 30]]
```

[15]:
```
# Dimensional operations
x = tf.random.normal((2, 3))
print(f"values: \n{x}")
y = tf.reduce_sum(x, axis=0) # sum over columns
print(f"values: \n{y}")
z = tf.reduce_sum(x, axis=1) # sum over rows
print(f"values: \n{z}")
```

```
values:
[[ 0.9868413   0.57056284  0.17946035]
 [ 0.83900064  1.0045967  -0.0642297 ]]
values:
[1.8258419  1.5751595  0.11523065]
values:
[1.7368646 1.7793677]
```

Indexing

Now we will look at how to extract, separate, and join values from tensors.

[18]:
```
x = tf.random.normal((3, 4))
print (f"x: \n{x}")
print()
print(f"x[:1]: \n{x[0]}")
print()
print(f"x[:1, 1:3]: \n{x[:1, 1:3]}")
```

```
x:
[[-0.8340743  -1.0207754  -0.6342568   0.51541275]
 [-0.590669    1.6425287   0.60286343 -0.61301523]
 [ 0.6574386   0.16823037  1.4946445   0.8306155 ]]

x[:1]:
[-0.8340743  -1.0207754  -0.6342568   0.51541275]

x[:1, 1:3]:
[[-1.0207754 -0.6342568]]
```

Slicing

[20]:
```
# Select with dimensional indices
x = tf.random.normal((2, 3))
print(f"values: \n{x}")

col_indices = tf.constant([0, 2])
chosen = tf.gather(x, axis=1, indices=col_indices) # values from column 0 & 2
print(f"values: \n{chosen}")

row_indices = tf.constant([0, 1])
col_indices = tf.constant([0, 2])
chosen = tf.gather_nd(x, indices=[row_indices, col_indices]) # values from (0, 0) & (1,␣
↪2)
print(f"values: \n{chosen}")
```

```
values:
[[-0.92759573  0.06353194  0.26116046]
 [-1.4687079  -0.90832067 -0.87465733]]
values:
[[-0.92759573  0.26116046]
 [-1.4687079  -0.87465733]]
values:
[0.06353194 0.26116046]
```

Joining

[21]:
```
# Concatenation
x = tf.random.normal((2, 3))
print(f"Values: \n{x}")
y = tf.concat([x, x], axis=0) # stack by rows (dim=1 to stack by columns)
print(f"Values: \n{y}")
```

```
Values:
[[ 2.260564   -0.6558232  -0.46591297]
 [ 0.4690215   1.026158   -0.11631647]]
Values:
[[ 2.260564   -0.6558232  -0.46591297]
 [ 0.4690215   1.026158   -0.11631647]
 [ 2.260564   -0.6558232  -0.46591297]
 [ 0.4690215   1.026158   -0.11631647]]
```

Gradients

- $y = 3x + 2$
- $z = \sum y/N$
- $\frac{\partial(z)}{\partial(x)} = \frac{\partial(z)}{\partial(y)}\frac{\partial(y)}{\partial(x)} = \frac{1}{N} * 3 = \frac{1}{12} * 3 = 0.25$

[22]:
```
# Tensors with gradient book keeping
x = tf.random.normal((3, 4))

# Tensorflow needs graph context to track gradients
with tf.GradientTape() as g:
    g.watch(x)
    y = 3*x + 2
    z = tf.reduce_mean(y)

dz_dx = g.gradient(z, x)

print(dz_dx)
```

```
tf.Tensor(
[[0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]
 [0.25 0.25 0.25 0.25]], shape=(3, 4), dtype=float32)
```

CUDA

This section details how to check if we are able to use GPU to accelerate our machine learning or deep learning models.

The Compute Unified Device Architecture or **CUDA** is a parallel computing platform and API that allows software to use certain types of GPUs for general purpose processing. It is an extension of the C and C++ programming languages.

`TensorFlow` makes using the GPU quite transparent. If the GPU compatible version of TF is installed along with the proper drivers, TF will use the GPU.

Although training is usually faster on the GPU, depending on model size and hardware specs, it can take quite a while to copy the model and your data to the GPU.

Link to metapackage for easily installing `TensorFlow` GPU using a conda 'metapackage'. This is just a special package which installs the required GPU drivers alongside TensorFlow.

[23]:
```
# Is CUDA available?
print(tf.test.is_built_with_cuda())
```

```
False
```

[26]:
```
# Set device to first gpu (if available)
device = "/gpu:0" if tf.test.is_built_with_cuda() else "cpu"
print(device)
```

```
cpu
```

[27]:
```
with tf.device(device):
    a = tf.constant([1, 2, 3])
```

[5]:
```
# Print info about local cpu/gpu devices through tensorflow library
from tensorflow.python.client import device_lib

# The most useful information is found in the first two lines of the output.
# 1st line is name of device (cpu/gpu and number).
# If there were 2 cpus on the device, there would be another entry in the list under␣
↪name: '/device:cpu:1'
# The second line give the memory limit in bits.
device_lib.list_local_devices()
```

[5]:
```
[name: "/device:CPU:0"
 device_type: "CPU"
 memory_limit: 268435456
 locality {
 }
 incarnation: 3513264468343507029
 xla_global_id: -1]
```

BACK TO TOP

7.36 Tutorial 9 - PyTorch

This tutorial is adapted from a 2022 blog post on the website Made With ML, by Goku Mohandas.

In this notebook, we will learn the basics of PyTorch, which is a Machine Learning library used to build dynamic neural networks. We'll learn about the basics, such as creating and using Tensors.

[1]:
```
import numpy as np
import torch
```

[2]:
```
SEED = 1234
```

[3]:
```
# Set seed for reproducibility
np.random.seed(seed=SEED)
torch.manual_seed(SEED)
```

[3]:
```
<torch._C.Generator at 0x7e1d7c9ebe10>
```

7.36.1 Basics

[4]:
```
# Creating a random tensor
x = torch.randn((2, 3))  # (rand(2,3) -> normal distribution)
print(f"Type: {x.type()}")
print(f"Size: {x.shape}")
print(f"Values: \n{x}")
```

```
Type: torch.FloatTensor
Size: torch.Size([2, 3])
Values:
tensor([[ 0.0461,  0.4024, -1.0115],
        [ 0.2167, -0.6123,  0.5036]])
```

[5]:
```
# Zeros and Ones tensor
x = torch.zeros(2, 3)
print (x)
x = torch.ones(2, 3)
print (x)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

[6]:
```
# List --> Tensor
x = torch.Tensor([[1, 2, 3],[4, 5, 6]])
print(f"Size: {x.shape}")
print(f"Values: \n{x}")
```

```
Size: torch.Size([2, 3])
Values:
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

[7]:
```
# NumPy array --> Tensor
x = torch.Tensor(np.random.rand(2, 3))
print(f"Size: {x.shape}")
print(f"Values: \n{x}")
```

```
Size: torch.Size([2, 3])
Values:
tensor([[0.1915, 0.6221, 0.4377],
        [0.7854, 0.7800, 0.2726]])
```

[8]:
```
# Changing tensor type
x = torch.Tensor(3, 4)
print(f"Type: {x.type()}")
x = x.long()
print(f"Type: {x.type()}")
```

```
Type: torch.FloatTensor
Type: torch.LongTensor
```

7.36.2 Operations

[9]:
```
# Addition
x = torch.randn(2, 3)
y = torch.randn(2, 3)
z = x + y
print(f"Size: {z.shape}")
print(f"Values: \n{z}")
```

```
Size: torch.Size([2, 3])
Values:
tensor([[ 0.0761, -0.6775, -0.3988],
        [ 3.0633, -0.1589,  0.3514]])
```

[10]:
```
# Dot product
x = torch.randn(2, 3)
y = torch.randn(3, 2)
z = torch.mm(x, y)
print(f"Size: {z.shape}")
print(f"Values: \n{z}")
```

```
Size: torch.Size([2, 2])
Values:
tensor([[ 1.0796, -0.0759],
        [ 1.2746, -0.5134]])
```

[11]:
```
# Transpose
x = torch.randn(2, 3)
print(f"Size: {x.shape}")
print(f"Values: \n{x}")
y = torch.t(x)
print(f"Size: {y.shape}")
print(f"Values: \n{y}")
```

```
Size: torch.Size([2, 3])
Values:
tensor([[ 0.8042, -0.1383,  0.3196],
        [-1.0187, -1.3147,  2.5228]])
Size: torch.Size([3, 2])
Values:
tensor([[ 0.8042, -1.0187],
        [-0.1383, -1.3147],
        [ 0.3196,  2.5228]])
```

[12]:
```
# Reshape
x = torch.randn(2, 3)
z = x.view(3, 2)
print(f"Size: {z.shape}")
print(f"Values: \n{z}")
```

```
Size: torch.Size([3, 2])
Values:
tensor([[ 0.4501,  0.2709],
        [-0.8087, -0.0217],
        [-1.0413,  0.0702]])
```

[13]:
```
# Dangers of reshaping (unintended consequences)
x = torch.tensor([
    [[1,1,1,1], [2,2,2,2], [3,3,3,3]],
    [[10,10,10,10], [20,20,20,20], [30,30,30,30]]
])
print(f"Size: {x.shape}")
print(f"x: \n{x}\n")

a = x.view(x.size(1), -1)
print(f"\nSize: {a.shape}")
print(f"a: \n{a}\n")

b = x.transpose(0,1).contiguous()
print(f"\nSize: {b.shape}")
print(f"b: \n{b}\n")

c = b.view(b.size(0), -1)
print(f"\nSize: {c.shape}")
print(f"c: \n{c}")
```

```
Size: torch.Size([2, 3, 4])
x:
tensor([[[ 1,  1,  1,  1],
         [ 2,  2,  2,  2],
         [ 3,  3,  3,  3]],

        [[10, 10, 10, 10],
         [20, 20, 20, 20],
         [30, 30, 30, 30]]])


Size: torch.Size([3, 8])
a:
tensor([[ 1,  1,  1,  1,  2,  2,  2,  2],
        [ 3,  3,  3,  3, 10, 10, 10, 10],
        [20, 20, 20, 20, 30, 30, 30, 30]])


Size: torch.Size([3, 2, 4])
b:
tensor([[[ 1,  1,  1,  1],
         [10, 10, 10, 10]],

        [[ 2,  2,  2,  2],
         [20, 20, 20, 20]],

        [[ 3,  3,  3,  3],
```

(continues on next page)

(continued from previous page)

```
         [30, 30, 30, 30]]])


Size: torch.Size([3, 8])
c:
tensor([[ 1,  1,  1,  1, 10, 10, 10, 10],
        [ 2,  2,  2,  2, 20, 20, 20, 20],
        [ 3,  3,  3,  3, 30, 30, 30, 30]])
```

[14]:
```
# Dimensional operations
x = torch.randn(2, 3)
print(f"Values: \n{x}")
y = torch.sum(x, dim=0) # add each row's value for every column
print(f"Values: \n{y}")
z = torch.sum(x, dim=1) # add each column's value for every row
print(f"Values: \n{z}")
```

```
Values:
tensor([[ 0.5797, -0.0599,  0.1816],
        [-0.6797, -0.2567, -1.8189]])
Values:
tensor([-0.1000, -0.3166, -1.6373])
Values:
tensor([ 0.7013, -2.7553])
```

Indexing, Slicing, and Joining

[15]:
```
x = torch.randn(3, 4)
print (f"x: \n{x}")
print (f"x[:1]: \n{x[:1]}")
print (f"x[:1, 1:3]: \n{x[:1, 1:3]}")
```

```
x:
tensor([[ 0.2111,  0.3372,  0.6638,  1.0397],
        [ 1.8434,  0.6588, -0.2349, -0.0306],
        [ 1.7462, -0.0722, -1.6794, -1.7010]])
x[:1]:
tensor([[0.2111, 0.3372, 0.6638, 1.0397]])
x[:1, 1:3]:
tensor([[0.3372, 0.6638]])
```

[16]:
```
# Select with dimensional indices
x = torch.randn(2, 3)
print(f"Values: \n{x}")

col_indices = torch.LongTensor([0, 2])
chosen = torch.index_select(x, dim=1, index=col_indices) # values from column 0 & 2
print(f"Values: \n{chosen}")

row_indices = torch.LongTensor([0, 1])
col_indices = torch.LongTensor([0, 2])
```

(continues on next page)

(continued from previous page)

```
chosen = x[row_indices, col_indices] # values from (0, 0) & (1, 2)
print(f"Values: \n{chosen}")
```

```
Values:
tensor([[ 0.6486,  1.7653,  1.0812],
        [ 1.2436,  0.8971, -0.0784]])
Values:
tensor([[ 0.6486,  1.0812],
        [ 1.2436, -0.0784]])
Values:
tensor([ 0.6486, -0.0784])
```

[17]:
```
# Concatenation
x = torch.randn(2, 3)
print(f"Values: \n{x}")
y = torch.cat([x, x], dim=0) # stack by rows (dim=1 to stack by columns)
print(f"Values: \n{y}")
```

```
Values:
tensor([[ 0.5548, -0.0845,  0.5903],
        [-1.0032, -1.7873,  0.0538]])
Values:
tensor([[ 0.5548, -0.0845,  0.5903],
        [-1.0032, -1.7873,  0.0538],
        [ 0.5548, -0.0845,  0.5903],
        [-1.0032, -1.7873,  0.0538]])
```

Gradients

- $y = 3x + 2$
- $z = \sum y/N$
- $\frac{\partial(z)}{\partial(x)} = \frac{\partial(z)}{\partial(y)}\frac{\partial(y)}{\partial(x)} = \frac{1}{N} * 3 = \frac{1}{12} * 3 = 0.25$

[18]:
```
# Tensors with gradient bookkeeping
x = torch.rand(3, 4, requires_grad=True)
y = 3*x + 2
z = y.mean()
z.backward() # z has to be scalar
print(f"x: \n{x}")
print(f"x.grad: \n{x.grad}")
```

```
x:
tensor([[0.7379, 0.0846, 0.4245, 0.9778],
        [0.6800, 0.3151, 0.3911, 0.8943],
        [0.6889, 0.8389, 0.1780, 0.6442]], requires_grad=True)
x.grad:
tensor([[0.2500, 0.2500, 0.2500, 0.2500],
        [0.2500, 0.2500, 0.2500, 0.2500],
        [0.2500, 0.2500, 0.2500, 0.2500]])
```

7.36.3 CUDA tensors

[19]:
```
# Is CUDA available?
print (torch.cuda.is_available())
```

```
True
```

If False, CUDA is not available. In Google Colab, we can change it by *Runtime* > *Change runtime type* > Change *Hardware accelerator* to *GPU* > Click *Save*

[20]:
```
# Is CUDA available now?
print (torch.cuda.is_available())
```

```
True
```

[21]:
```
# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print (device)
```

```
cuda
```

[22]:
```
x = torch.rand(2,3)
print (x.is_cuda)
x = torch.rand(2,3).to(device) # Tensor is stored on the GPU
print (x.is_cuda)
```

```
False
True
```

BACK TO TOP

7.37 Tutorial 10 - Tensorflow Datasets

Public datasets are an important resource for accelerating machine learning research. However, writing custom scripts to fetch and prepare each dataset individually can be tedious.

TensorFlow DataSets (TFDS) handles the tasks of sourcing the data and standardizing it into a consistent format. Furthermore, TFDS utilizes the `tensorflow.data API` to construct high-performance input pipelines that are seamlessly usable with tensorflow.keras models.

7.37.1 Overview

TFDS is a set of ready-to-use datasets for various machine learning tasks, including Computer Vision datasets, Natural Language Processing datasets, and miscellaneous other datasets for performing Unsupervised Learning, Reinforcement learning, and more.

The entire list of available datasets can be found here.

All of these datasets are contained under the `tensorflow.data.Datasets` module.

To install TFDS:

```
pip install tfds-nightly
```

TFDS is pre-installed in Google Colab, and it can be directly imported as in the next cell.

[1]:
```
import tensorflow as tf
import tensorflow_datasets as tfds
import numpy as np
```

The following line also displays the list all the available datasets in TFDS.

[ ]:
```
tfds.list_builders()
```

7.37.2 Load Dataset with TFDS

The easiest way of loading a dataset with TFDS is with `tfds.load`.

It will:

1. Download the data and save it as `tfrecord` files.
2. Load the `tfrecord` and create the `tf.data.Dataset`.

[3]:
```
(training_data, test_data), info = tfds.load('mnist', with_info=True, shuffle_files=True,
↪ as_supervised=True, split=['train', 'test'])
```

Arguments in `tfds.load` include:

- First argument is the name of dataset.
- param `'split'` controls which split we wish to load (e.g., train, test, or validation).
- param `'shuffle_files'` controls whether or not data is shuffled between each epoch.
- param `'data_dir'` controls where the dataset is saved (defaults to `~/tensorflow_datasets/`).
- param `'with_info'` controls whether or not the metadata for the dataset is included.
- param `'as_supervised'` controls whether or not a tuple `(features, label)` is returned (as opposed to just features).
- param `'download'` controls whether or not the library will attempt to download the dataset.

We can access the dataset metadata with `info` as in the next cell.

[4]:
```
print(info)
```

```
tfds.core.DatasetInfo(
    name='mnist',
    full_name='mnist/3.0.1',
    description="""
    The MNIST database of handwritten digits.
    """,
    homepage='http://yann.lecun.com/exdb/mnist/',
    data_dir='C:\\Users\\vakanski\\tensorflow_datasets\\mnist\\3.0.1',
    file_format=tfrecord,
    download_size=11.06 MiB,
    dataset_size=21.00 MiB,
    features=FeaturesDict({
```

(continues on next page)

(continued from previous page)

```
        'image': Image(shape=(28, 28, 1), dtype=uint8),
        'label': ClassLabel(shape=(), dtype=int64, num_classes=10),
    }),
    supervised_keys=('image', 'label'),
    disable_shuffling=False,
    splits={
        'test': <SplitInfo num_examples=10000, num_shards=1>,
        'train': <SplitInfo num_examples=60000, num_shards=1>,
    },
    citation="""@article{lecun2010mnist,
      title={MNIST handwritten digit database},
      author={LeCun, Yann and Cortes, Corinna and Burges, CJ},
      journal={ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist},
      volume={2},
      year={2010}
    }""",
)
```

Features metadata can inlcude features shape, label shape, etc.

[5]:
```
info.features
```

[5]:
```
FeaturesDict({
    'image': Image(shape=(28, 28, 1), dtype=uint8),
    'label': ClassLabel(shape=(), dtype=int64, num_classes=10),
})
```

We can also inspect the number of classes and label names.

[6]:
```
print(info.features["label"].num_classes)
print(info.features["label"].names)
```

```
10
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

The following code prints shapes and dtypes of the data.

[7]:
```
print(info.features.shape)
print(info.features.np_dtype)
print(info.features['image'].shape)
print(info.features['image'].np_dtype)
```

```
{'image': (28, 28, 1), 'label': ()}
{'image': <class 'numpy.uint8'>, 'label': <class 'numpy.int64'>}
(28, 28, 1)
<class 'numpy.uint8'>
```

Slicing API for Customized Dataset Split

[8]:
```
# Fashion MNIST dataset, get 25% to 75% of train dataset
ds = tfds.load('mnist', split='train[25%:75%]')
```

[9]:
```
# Get first 4,000 of the data for training
ds = tfds.load('fashion_mnist', split='train[:4000]')
```

[10]:
```
# Get 25% of training and all of the test data
ds = tfds.load('fashion_mnist', split='train[:25%]+test')
```

Cross-Validation

[11]:
```
vals_ds = tfds.load('fashion_mnist', split=[
    f'train[{k}%:{k+10}%]' for k in range(0, 100, 10)])

trains_ds = tfds.load('fashion_mnist', split=[
    f'train[:{k}%]+train[{k+10}%:]' for k in range(0, 100, 10)])
```

[12]:
```
# First fold of validation DS
vals_ds_fold1 =vals_ds[0]
```

Iterate over the Dataset in Batches

[13]:
```
batch_size = 32

for epoch in range(1):
    print(f'epoch {epoch}')
    n = 0
    for img, label in training_data.batch(batch_size):
        # print the first 10 batches
        while n < 10:
            # notice that img.shape = [batch size, pixels width, pixels heights,␣
↪channels number)
            # notice that label.shape = 32, therefore 32 labels are shown
            print(f'img: {img.shape}, labels: {label}')
            # we can train a model here
            n = n+1
```

```
epoch 0
img: (32, 28, 28, 1), labels: [4 1 0 7 8 1 2 7 1 6 6 4 7 7 3 3 7 9 9 1 0 6 6 9 9 4 8 9 4␣
↪7 3 3]
img: (32, 28, 28, 1), labels: [4 1 0 7 8 1 2 7 1 6 6 4 7 7 3 3 7 9 9 1 0 6 6 9 9 4 8 9 4␣
↪7 3 3]
img: (32, 28, 28, 1), labels: [4 1 0 7 8 1 2 7 1 6 6 4 7 7 3 3 7 9 9 1 0 6 6 9 9 4 8 9 4␣
↪7 3 3]
img: (32, 28, 28, 1), labels: [4 1 0 7 8 1 2 7 1 6 6 4 7 7 3 3 7 9 9 1 0 6 6 9 9 4 8 9 4␣
↪7 3 3]
img: (32, 28, 28, 1), labels: [4 1 0 7 8 1 2 7 1 6 6 4 7 7 3 3 7 9 9 1 0 6 6 9 9 4 8 9 4␣
↪7 3 3]
```

(continues on next page)

(continued from previous page)

```
img: (32, 28, 28, 1), labels: [4 1 0 7 8 1 2 7 1 6 6 4 7 7 3 3 7 9 9 1 0 6 6 9 9 4 8 9 4␣
↪7 3 3]
img: (32, 28, 28, 1), labels: [4 1 0 7 8 1 2 7 1 6 6 4 7 7 3 3 7 9 9 1 0 6 6 9 9 4 8 9 4␣
↪7 3 3]
img: (32, 28, 28, 1), labels: [4 1 0 7 8 1 2 7 1 6 6 4 7 7 3 3 7 9 9 1 0 6 6 9 9 4 8 9 4␣
↪7 3 3]
img: (32, 28, 28, 1), labels: [4 1 0 7 8 1 2 7 1 6 6 4 7 7 3 3 7 9 9 1 0 6 6 9 9 4 8 9 4␣
↪7 3 3]
img: (32, 28, 28, 1), labels: [4 1 0 7 8 1 2 7 1 6 6 4 7 7 3 3 7 9 9 1 0 6 6 9 9 4 8 9 4␣
↪7 3 3]
```

Visualization

Method1: features and label samples from TFDS objects can be visualized in a Jupyter Notebook by using `tfds.as_dataframe` to convert into pandas DataFrame.

[14]:
```
# list 5 images and labels from mnist dataset
ds, info = tfds.load('mnist', split='train', with_info=True)

tfds.as_dataframe(ds.take(5), info)
```

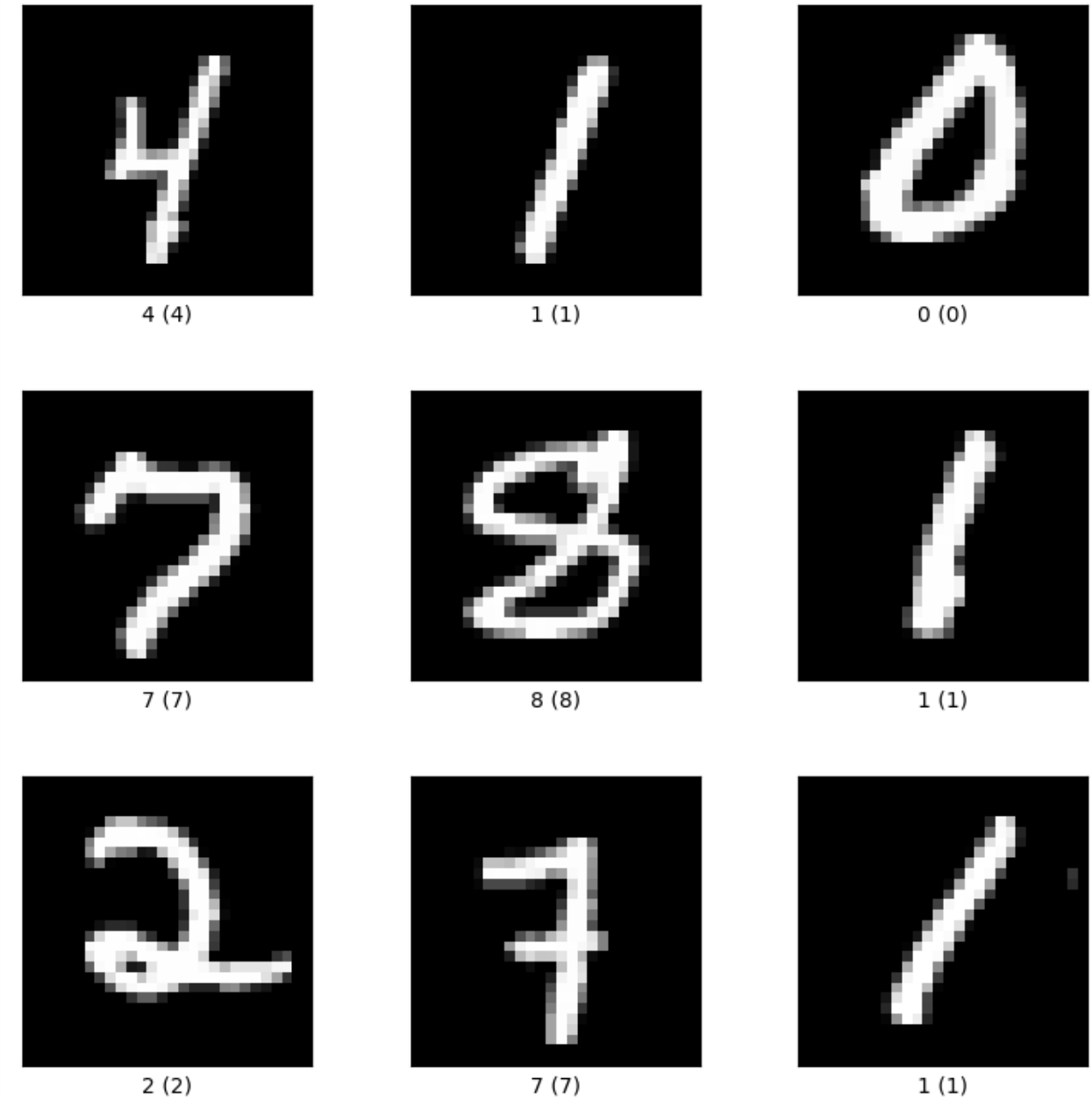
[14]:
```
                                               image  label
0  [[[0], [0], [0], [0], [0], [0], [0], [0], [0],...      4
1  [[[0], [0], [0], [0], [0], [0], [0], [0], [0],...      1
2  [[[0], [0], [0], [0], [0], [0], [0], [0], [0],...      0
3  [[[0], [0], [0], [0], [0], [0], [0], [0], [0],...      7
4  [[[0], [0], [0], [0], [0], [0], [0], [0], [0],...      8
```

[15]:
```
# list 5 images and labels from mnist dataset
tfds.as_dataframe(ds.take(5), info)
```

[15]:
```
                                               image  label
0  [[[0], [0], [0], [0], [0], [0], [0], [0], [0],...      4
1  [[[0], [0], [0], [0], [0], [0], [0], [0], [0],...      1
2  [[[0], [0], [0], [0], [0], [0], [0], [0], [0],...      0
3  [[[0], [0], [0], [0], [0], [0], [0], [0], [0],...      7
4  [[[0], [0], [0], [0], [0], [0], [0], [0], [0],...      8
```

Method 2: `tfds.show_examples` returns a matplotlib figure with data samples and labels (only image datasets are supported with this method).

[16]:
```
import matplotlib.pyplot as plt

fig = tfds.show_examples(ds, info)
```



Create Your Own TFDS with `tf.data.Dataset.from_tensor_slices`

[17]:
```
np.random.seed(1)

features = np.random.uniform(0,1, size=(32, 100, 100))
labels = np.random.randint(0, 2, size=(32, 1))

dataset = tf.data.Dataset.from_tensor_slices((features, labels))
dataset = dataset.shuffle(buffer_size=1024).batch(5)
```

(continues on next page)

(continued from previous page)

```
for epoch in range(2):
    for step, (x_batch, y_batch) in enumerate(dataset):
        print(f'step: {step}, x_batch shape: {x_batch.shape}, y_batch shape: {y_batch.
↪shape}')
    print()
```

```
step: 0, x_batch shape: (5, 100, 100), y_batch shape: (5, 1)
step: 1, x_batch shape: (5, 100, 100), y_batch shape: (5, 1)
step: 2, x_batch shape: (5, 100, 100), y_batch shape: (5, 1)
step: 3, x_batch shape: (5, 100, 100), y_batch shape: (5, 1)
step: 4, x_batch shape: (5, 100, 100), y_batch shape: (5, 1)
step: 5, x_batch shape: (5, 100, 100), y_batch shape: (5, 1)
step: 6, x_batch shape: (2, 100, 100), y_batch shape: (2, 1)

step: 0, x_batch shape: (5, 100, 100), y_batch shape: (5, 1)
step: 1, x_batch shape: (5, 100, 100), y_batch shape: (5, 1)
step: 2, x_batch shape: (5, 100, 100), y_batch shape: (5, 1)
step: 3, x_batch shape: (5, 100, 100), y_batch shape: (5, 1)
step: 4, x_batch shape: (5, 100, 100), y_batch shape: (5, 1)
step: 5, x_batch shape: (5, 100, 100), y_batch shape: (5, 1)
step: 6, x_batch shape: (2, 100, 100), y_batch shape: (2, 1)
```

Simple Example of a Pipeline with TFDS in a CNN Model

[18]:
```
import tensorflow as tf
import tensorflow_datasets as tfds

# MODEL DEFINITION
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16, (3,3), activation='relu', input_shape=(300, 300, 3)),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(12, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer='Adam', loss='binary_crossentropy', metrics=['accuracy'])

# EXTRACT PHASE
data = tfds.load('horses_or_humans', split='train', as_supervised=True)
val_data = tfds.load('horses_or_humans', split='test', as_supervised=True)

# TRANSFORM PHASE
def augmentimages(image, label):
  image = tf.cast(image, tf.float32)
  image = (image/255)
  image = tf.image.random_flip_left_right(image)
  return image, label

train = data.map(augmentimages)
train_batches = train.shuffle(100).batch(32)
```

(continues on next page)

(continued from previous page)

```
validation_batches = val_data.batch(32)

# MODEL TRAINING PHASE
history = model.fit(train_batches, epochs=3,
                    validation_data=validation_batches, validation_steps=1)
```

```
Epoch 1/3
33/33 [==============================] - 18s 509ms/step - loss: 3.5113 - accuracy: 0.
↪5083 - val_loss: 0.6937 - val_accuracy: 0.4375
Epoch 2/3
33/33 [==============================] - 15s 435ms/step - loss: 0.6930 - accuracy: 0.
↪5131 - val_loss: 0.6940 - val_accuracy: 0.4375
Epoch 3/3
33/33 [==============================] - 13s 401ms/step - loss: 0.6930 - accuracy: 0.
↪5131 - val_loss: 0.6940 - val_accuracy: 0.4375
```

References:

1. Moroney, L. (n.d.). AI and Machine Learning for Coders. O'Reilly Online Learning. https://www.oreilly.com/library/view/ai-and-machine/9781492078180/ch04.html
2. Tensorflow datasets. TensorFlow. (n.d.). https://www.tensorflow.org/datasets/overview#iterate_over_a_dataset

BACK TO TOP

7.38 Tutorial 11 - CometML

This tutorial is adapted from a blog post titled "Getting Started with Comet ML" by Angelica Lo Duca.

7.38.1 Overview

CometML is an online experimentation platform for testing Machine Learning projects (similar to Neptune.ai, Guild.ai, etc.). Its main advantage is that it makes it very easy to build reporting dashboards and monitor Machine Learning projects.

CometML can be easily integrated with most popular ML libraries like scikit-learn, Keras, and others. The experiments can be written in Python, Javascript, Java, R, and REST APIs. In this tutorial we will only be discussing how to use the Python SDK, but the SDKs for other languages should be similar enough.

Features of CometML

- Users can easily build and compare the results of different experiments for the same project.
- The model can be monitored from the early stages up to debugging.
- Makes collaborating with other project devs easy (note that this feature is not included with the free account).
- Easily build reports and panels.
- Share projects publicly through their platform.

7.38.2 Working with CometML

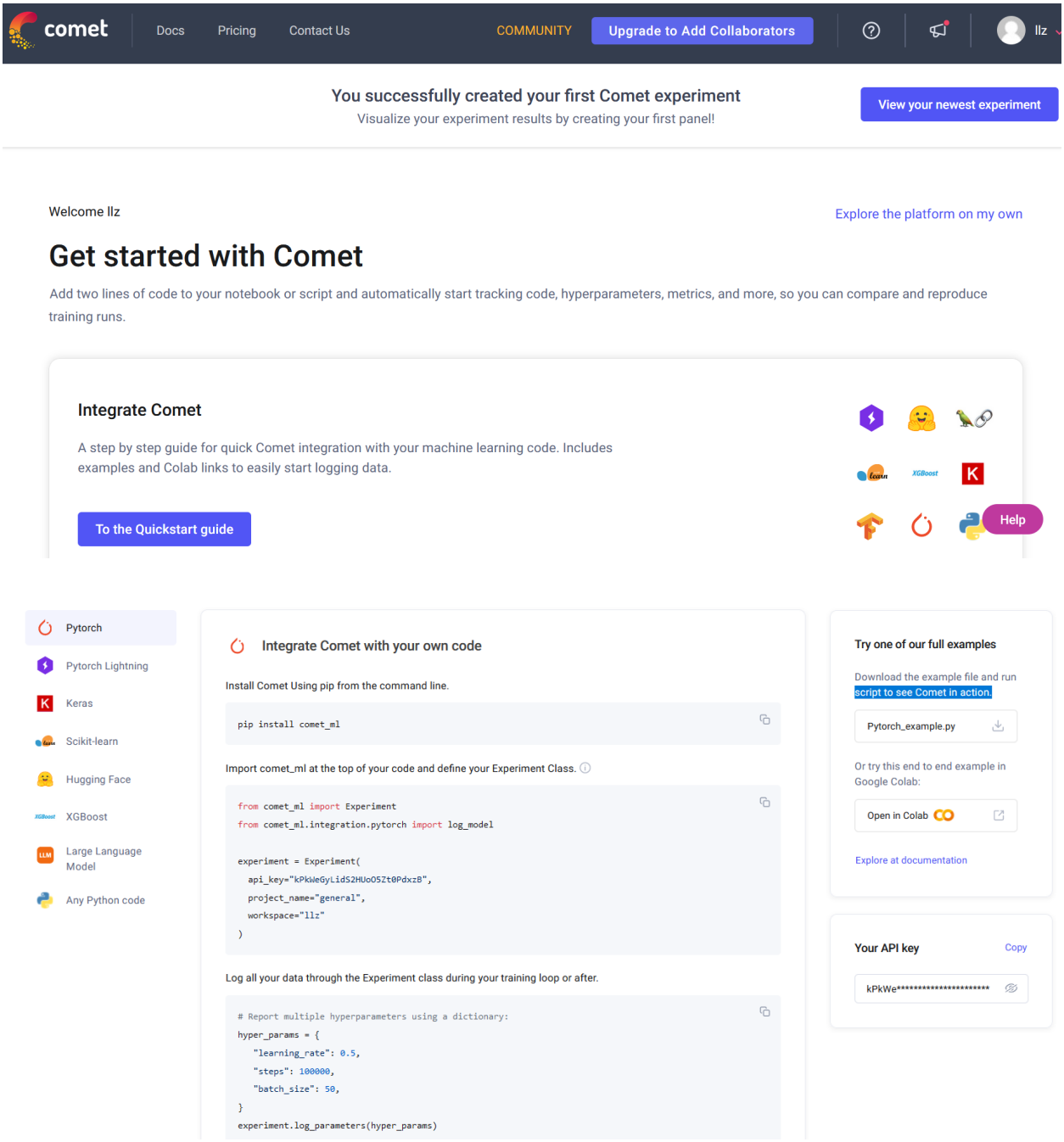
Step 1: Create a free account, log in, and create a new project.

Step 2: After login with your account, click "To the Quickstart Guide".

Step 3: The webpage will then generate a snippet of code that is to be placed in the project you are working on (shown in the figure below).

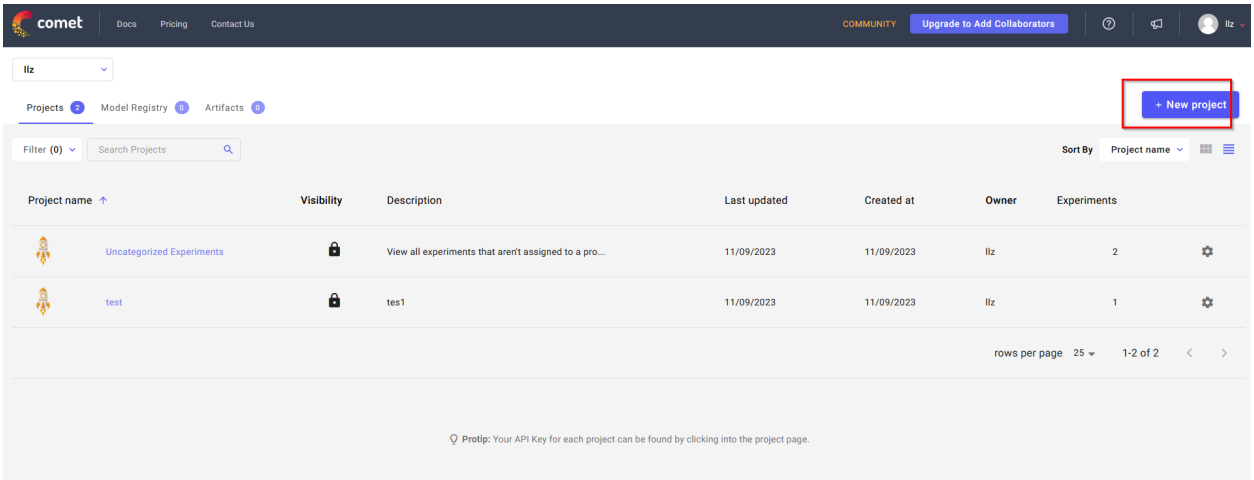
Step 4: Install the Comet ML library using `pip install comet_ml`

Step 5: Copy and paste the second block of codes shown below in your Python project.



The default "Project name" is "Uncategorized Experiments" as shown in the following figure from the CometML

webpage. You can create a `New project` and assign it another name on the CometML webpage, and then all you need to do is change the "project_name" variable to the new project name.



`Experiment` is the core class of CometML. An Experiment represents a unit of measurable research that defines a single execution of code with some associated data; for example, training a model using a single set of hyperparameters. Use Experiment to log new data to the CometML UI.

An Experiment automatically logs scripts output (stdout/stderr), code, and command-line arguments on any script and for the supported libraries, also logs hyperparameters, metrics, and model configuration.

[ ]:
```
from comet_ml import Experiment
from comet_ml.integration.pytorch import log_model

experiment = Experiment(
  api_key="kPkWeGyLidS2HUoO5Zt0PdxzB",
  project_name="general",
  workspace="llz"
)
```

Example usage

To show how to use the library, let's consider an example project using the heart attack dataset from Kaggle. The task for this dataset is to predict whether or not a patient has a high chance of heart attack, given several features including age, sex, resting blood pressure, etc.

[10]:
```
import pandas as pd

# Load data:
df = pd.read_csv('data/heart.csv')
df.head()
```

[10]:

	age	sex	cp	trtbps	chol	fbs	restecg	thalachh	exng	oldpeak	slp	\
0	63	1	3	145	233	1	0	150	0	2.3	0	
1	37	1	2	130	250	0	1	187	0	3.5	0	
2	41	0	1	130	204	0	0	172	0	1.4	2	
3	56	1	1	120	236	0	1	178	0	0.8	2	
4	57	0	0	120	354	0	1	163	1	0.6	2	

(continues on next page)

(continued from previous page)

```
   caa  thall  output
0    0      1       1
1    0      2       1
2    0      2       1
3    0      2       1
4    0      2       1
```

[11]:
```
# separate data into features and targets
x = df.drop(columns=['output'])
y = df['output']
```

[ ]:
```
x.head()
```

```
   age  sex  cp  trtbps  chol  fbs  restecg  thalachh  exng  oldpeak  slp  \
0   63    1   3     145   233    1        0       150     0      2.3    0
1   37    1   2     130   250    0        1       187     0      3.5    0
2   41    0   1     130   204    0        0       172     0      1.4    2
3   56    1   1     120   236    0        1       178     0      0.8    2
4   57    0   0     120   354    0        1       163     1      0.6    2

   caa  thall
0    0      1
1    0      2
2    0      2
3    0      2
4    0      2
```

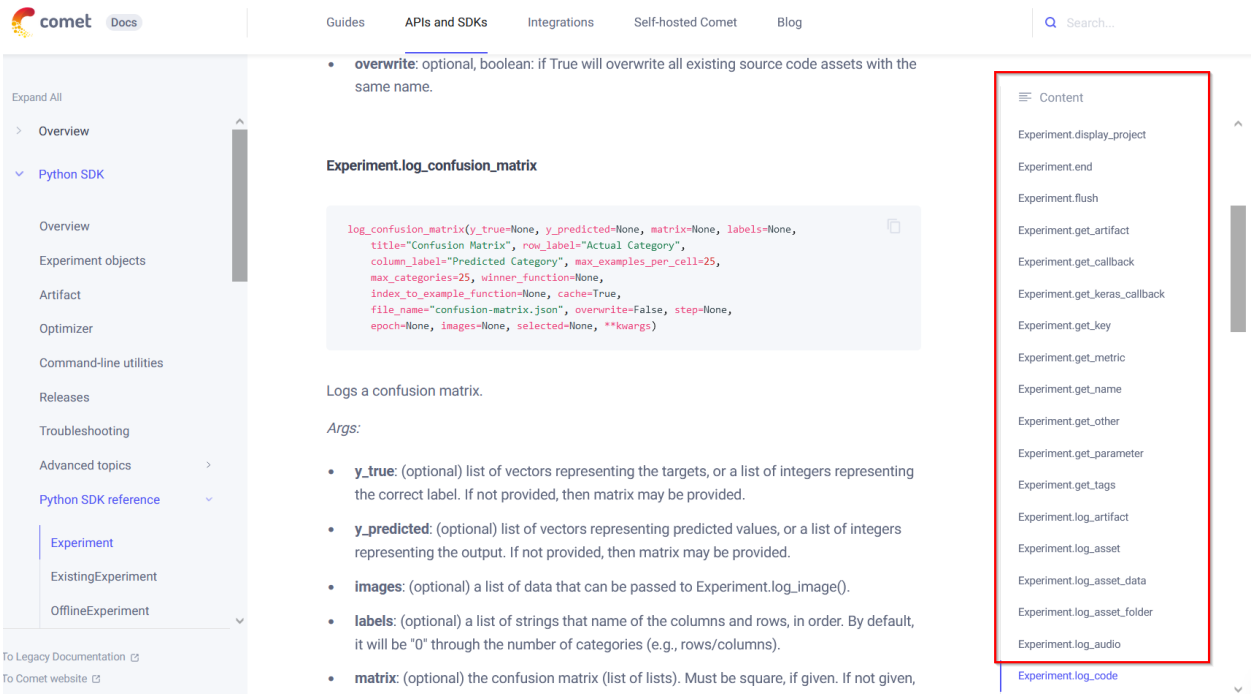
[12]:
```
y.head()
```

[12]:
```
0    1
1    1
2    1
3    1
4    1
Name: output, dtype: int64
```

[13]:
```
from sklearn.model_selection import train_test_split

# create train/test split
x_train, x_test, y_train, y_test = train_test_split(x, y, random_state=42)
```

[14]:
```
from sklearn.preprocessing import MinMaxScaler

# scale inputs
scaler = MinMaxScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)
```

We will show an example of logging confusion matrix information with CometML. There are numerous logging applications in Experiment Reference.



[15]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report
import numpy as np

# create model:
model = DecisionTreeClassifier(random_state=42, max_depth=2)
min_samples = 5

target_names = ['class 0', 'class 1']

# examine how the number of training samples affects model performance:
for step in range(min_samples, len(x_train)):

    model.fit(x_train[:step], y_train[:step])

    y_pred = model.predict(x_test)
    report = classification_report(y_test, y_pred, target_names=target_names, output_
↪dict=True)

    for label, metric in report.items():
        try:
            experiment.log_metrics(metric, prefix=label, step=step)

        except:
            experiment.log_metric(label, metric, step=step)

    experiment.log_confusion_matrix(y_test.tolist(), y_pred.tolist())

experiment.display(tab='confusion-matrices')
experiment.end()
```

```
<IPython.lib.display.IFrame at 0x7ef96b721ba0>
COMET INFO: ------------------------------------------------------------------------------
↪----------
COMET INFO: Comet.ml Experiment Summary
COMET INFO: ------------------------------------------------------------------------------
↪----------
COMET INFO:   Data:
COMET INFO:     display_summary_level : 1
COMET INFO:     url                   : https://www.comet.com/llz/general/
↪3076c8951afd4185a68717788a151ffb
COMET INFO:   Metrics [count] (min, max):
COMET INFO:     accuracy [222]               : (0.39473684210526316, 0.7631578947368421)
COMET INFO:     class 0_f1-score [222]       : (0.37837837837837845, 0.7272727272727272)
COMET INFO:     class 0_precision [222]      : (0.358974358974359, 0.8421052631578947)
COMET INFO:     class 0_recall [222]         : (0.34285714285714286, 0.9714285714285714)
COMET INFO:     class 0_support              : 35
COMET INFO:     class 1_f1-score [222]       : (0.046511627906976744, 0.7906976744186047)
COMET INFO:     class 1_precision [222]      : (0.43243243243243246, 0.8064516129032258)
COMET INFO:     class 1_recall [222]         : (0.024390243902439025, 0.926829268292683)
COMET INFO:     class 1_support              : 41
COMET INFO:     macro avg_f1-score [222]     : (0.33518241945807553, 0.7589852008456659)
COMET INFO:     macro avg_precision [222]    : (0.39570339570339574, 0.7648745519713261)
COMET INFO:     macro avg_recall [222]       : (0.3951219512195122, 0.7574912891986063)
COMET INFO:     macro avg_support            : 76
COMET INFO:     weighted avg_f1-score [222]  : (0.31239262012509406, 0.7614888171803716)
COMET INFO:     weighted avg_precision [222] : (0.3986030564977934, 0.7641388417279759)
COMET INFO:     weighted avg_recall [222]    : (0.39473684210526316, 0.7631578947368421)
COMET INFO:     weighted avg_support         : 76
COMET INFO:   Parameters:
COMET INFO:     ccp_alpha                : 0.0
COMET INFO:     class_weight             : 1
COMET INFO:     clip                     : False
COMET INFO:     copy                     : True
COMET INFO:     criterion                : gini
COMET INFO:     feature_range            : (0, 1)
COMET INFO:     max_depth                : 2
COMET INFO:     max_features             : 1
COMET INFO:     max_leaf_nodes           : 1
COMET INFO:     min_impurity_decrease    : 0.0
COMET INFO:     min_samples_leaf         : 1
COMET INFO:     min_samples_split        : 2
COMET INFO:     min_weight_fraction_leaf : 0.0
COMET INFO:     random_state             : 42
COMET INFO:     splitter                 : best
COMET INFO:   Uploads:
COMET INFO:     confusion-matrix    : 222
COMET INFO:     environment details : 1
COMET INFO:     filename            : 1
COMET INFO:     installed packages  : 1
COMET INFO:     notebook            : 2
COMET INFO:     os packages         : 1
COMET INFO:     source_code         : 1
```

(continues on next page)

(continued from previous page)

```
COMET INFO:
COMET INFO: Please wait for metadata to finish uploading (timeout is 3600 seconds)
COMET INFO: Uploading 3154 metrics, params and output messages
COMET INFO: Uploading 2036 metrics, params and output messages
```

Reference

1. Blank, D. (2023, November 9). Overview - Comet Docs. https://www.comet.com/docs/v2/api-and-sdk/python-sdk/reference/Experiment/

BACK TO TOP

7.39 Tutorial 12 - GitHub

7.39.1 Overview

Git is a distributed version control system (DVCS) that allows developers to track changes in their code over time. It was created by Linus Torvalds in 2005 and is widely used for managing source code for software projects. Git enables multiple developers to work on a project simultaneously and independently, and it tracks changes made by each contributor. It provides features such as branching, merging, and history tracking.

With Git, each developer has a local copy of the entire project history. Developers can work offline, commit changes locally, and then synchronize with a central repository later.

Key features of Git include:

- Manage projects with repositories
- Clone a project to work on a local copy
- Control and track changes with staging and committing
- Branch and merge for working on different parts and versions of a project
- Pull the latest version of the project to a local copy
- Push local updates to the main project

A **repository** is typically used to organize a single project. Repositories can contain folders and files, images, videos, spreadsheets, and data sets. I.e., they can contain anything your project needs. Repositories typically include a README file that provides information about the project. README files are written in plain text Markdown language.

GitHub is a web-based platform that provides hosting for Git repositories. It offers a graphical interface for managing Git repositories, collaboration features, and additional tools for project management. GitHub allows multiple developers to collaborate on a project, track issues, and manage pull requests.

Key features of GitHub are:

- Repository hosting: GitHub provides a place to store and manage Git repositories.
- Collaboration: Developers can collaborate on projects by forking repositories, making changes, and submitting pull requests.
- Issue Tracking: GitHub includes an issue-tracking system for managing bugs, feature requests, and other tasks.

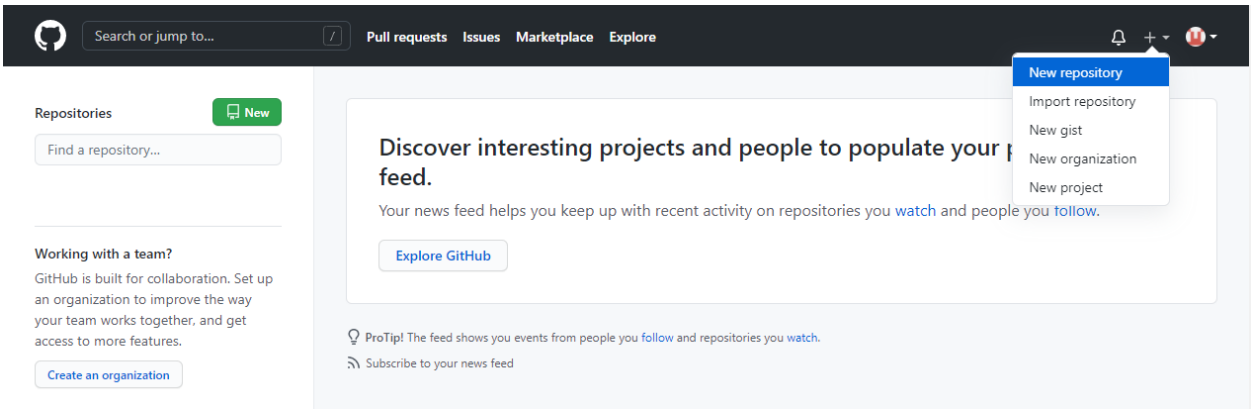
- Pull Requests: Developers can propose changes to a project by submitting pull requests, which can be reviewed and merged by project maintainers.
- Web-based interface: GitHub provides a user-friendly web interface for interacting with Git repositories.

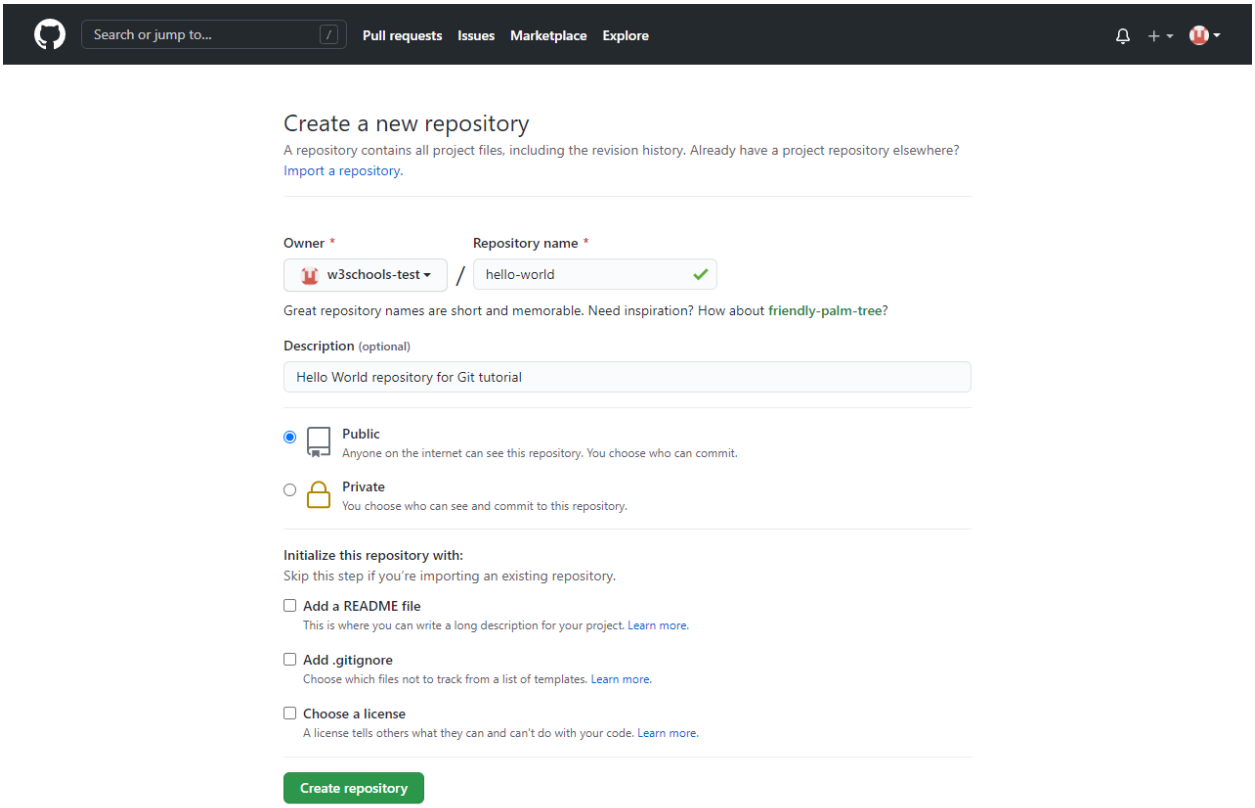
To summarize, Git serves as the version control system enabling developers to monitor code changes, whereas GitHub functions as a web-based platform offering hosting for Git repositories along with extra collaboration tools.

7.39.2 Working with GitHub

Step 1: Sign up for a GitHub account: link

Step 2: Create a Repository on GitHub by clicking on the "New repository" button shown below in the upper right corner. Then, fill in the relevant details. You can set the repository as "Public" or "Private". You also have options to add a README file, or choose a license for your repository. Click the "Create Repository" button to finish the creation process.

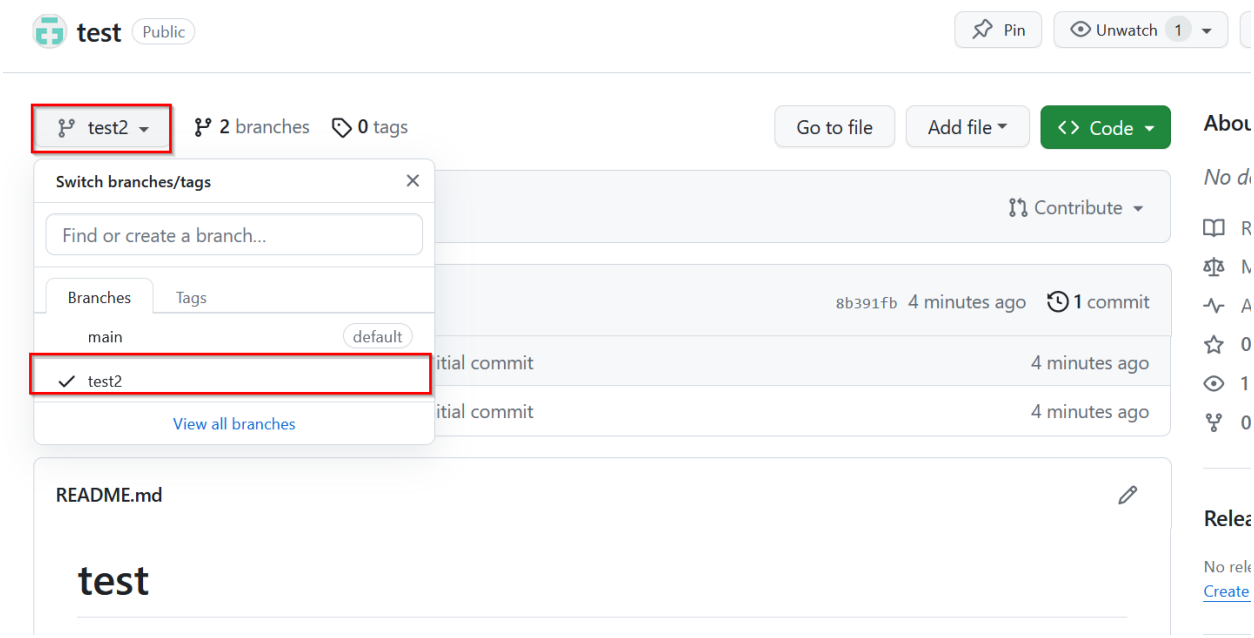




Step 3: Create a **branch**. By default, a repository comes with a primary branch called "`main`", which serves as the authoritative branch. We have the option to establish additional branches stemming from the main branch within the repository. The creation of branches allows for the simultaneous existence of various versions of a project. This proves beneficial when incorporating new features into a project without altering the primary source code. Changes made in different branches remain isolated from the main branch until a later stage when merging is addressed. Branches provide a space for experimentation and editing before finalizing changes and committing them to the main branch.

When branching off the main branch, we essentially create a duplicate or snapshot of the main branch as it existed at that specific moment. If someone else modifies the main branch while you are developing your branch, you have the option to incorporate those updates.

To create a new Branch, type in a descriptive name in the field below, and click Create Branch. In the following figure, the name of the new branch name is `test2`.



An example of a flow diagram for making changes to a branch is depicted next, showing the `'main'` branch, a new branch called `'feature'`, and the path of the `'feature'` branch before it is merged into the `'main'` branch.

Making and Committing Changes

We can make and save changes to the files in a repository. On GitHub, the saved changes are called **commits**. Each commit has an associated commit message, which is a description explaining why a particular change was made. Commit messages capture the history of the changes, so that other contributors can understand what you have done and why.

For instance, to edit the `README.md` file, we can follow these steps:

1) Under the branch we created, click the `README.md` file.
2) To edit the file, click on the Edit (pencil) button.
3) In the editor, write some text about the repository.
4) Click "Commit Changes…"
5) In the "Commit Changes" box, write a commit message that describes the changes.
6) Click "Commit Changes".

Opening a pull request

Pull requests are the main form of collaboration on GitHub. When you open a pull request, you are proposing your changes and requesting that someone review and pull in your contribution and merge them into their branch. Pull requests show the differences of the content from both branches. The changes, additions, and subtractions are shown in different colors.

As soon as you make a commit, you can open a pull request and start a discussion, even before the code is finished. By using GitHub's @mention feature in the pull request message, you can ask for feedback from specific people or teams, whether they are down the hall or 10 time zones away.

You can even open pull requests in your own repository and merge them yourself. It's a great way to learn the GitHub flow before working on larger projects.

To create a pull request:

1) Click the Pull Requests tab of the repository.
2) Click New pull request.
3) In the Example Comparisons box, select the branch you made to compare with the main (the original).
4) Look over your changes in the diffs on the Compare page, make sure they are what you want to submit.
5) Click Create Pull Request.
6) Give your pull request a title and write a brief description of your changes. You can include emojis and drag and drop images and gifs.
7) Optionally, to the right of your title and description, click the button next to Reviewers, Assignees, Labels, Projects, or Milestone to add any of these options to your pull request. You do not need to add any, but these options offer different ways to collaborate using pull requests.
8) Click Create Pull Request.

Merging a Pull Request

In this final step, you will merge your branch into the main branch. After you merge your pull request, the changes on your branch will be incorporated into main.

Sometimes, a pull request may introduce changes to code that conflict with the existing code on main. If there are any conflicts, GitHub will alert you about the conflicting code and prevent merging until the conflicts are resolved. You can make a commit that resolves the conflicts or use comments in the pull request to discuss the conflicts with your team members.

To merge your branch into the main branch:

1) At the bottom of the pull request, click Merge Pull Request to merge the changes into main.
2) Click Confirm Merge. You will receive a message that the request was successfully merged and the request was closed.
3) Click Delete Branch. Now that your pull request is merged and your changes are on main, you can safely delete the readme-edits branch. If you want to make more changes to your project, you can always create a new branch and repeat this process.

Working with a Local Git Repository

1) Download and install Git from link
2) Specify your Git information:

```
git config --global user.name "username"
git config --global user.email "youremailaddress
```

3) Create a local Git folder:

```
mkdir myproject
```

4) Initialize Git:

```
git init
```

Pull Request

5) Pull from Github with "Fetch" and "Merge":

```
git fetch origin
git merge origin/master
```

6) Simple Pull request (combination of Fetch and Merge):

```
git pull origin
```

Push your code from a Local folder to Github:

```
git push origin
```

References

1) Hello world. GitHub Docs. (n.d.). https://docs.github.com/en/get-started/quickstart/hello-world
2) H. F., Campion. An intro to Git and GitHub for beginners (tutorial). HubSpot Careers. https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners
3) Ajibola.Segunemmanuel. (2022, September 27). How to use Git and GitHub – Introduction for Beginners. freeCodeCamp.org. https://www.freecodecamp.org/news/introduction-to-git-and-github/
4) Git tutorial. (n.d.). https://www.w3schools.com/git/

BACK TO TOP