Introduction to Data Science with Python

Arvind Krishna, and Arend Kuyper

9/20/2022

Table of contents

Preface

This book is developed for the course STAT303-1 (Data Science with Python-1). The first two chapters of the book are a review of python, and will be covered very quickly. Students are expected to know the contents of these chapters beforehand, or be willing to learn it quickly. The core part of the course begins from the third chapter - *Reading data*.

Note that this book is still being edited. Please let the instructors know in case of any typos/mistakes/general feedback.

1 Introduction to Python and Jupyter Notebooks

This chapter is a very brief introduction to python and Jupyter notebooks. We only discuss the content relevant for applying python to analyze data.

Anaconda: If you are new to python, we recommend downloading the Anaconda installer and following the instructions for installation. Once installed, we'll use the Jupyter Notebook interface to write code.

Quarto: We'll use Quarto to publish the *.ipynb* file containing text, python code, and the output. Download and install Quarto from here.

1.1 Jupyter notebook

1.1.1 Introduction

Jupyter notebook is an interactive platform, where you can write code and text, and make visualizations. You can access Jupyter notebook from the Anaconda Navigator, or directly open the Jupyter Notebook application itself. It should automatically open up in your default browser. The figure below shows a Jupyter Notebook opened with Google Chrome. This page is called the *landing page* of the notebook.

```
<IPython.core.display.Image object>
```

To create a new notebook, click on the `New` button and select the `Python 3` option. You should see a blank notebook as in the figure below.

```
<IPython.core.display.Image object>
```

1.1.2 Writing and executing code

Code cell: By default, a cell is of type *Code*, i.e., for typing code, as seen as the default choice in the dropdown menu below the *Widgets* tab. Try typing a line of python code (say, `2+3`) in an empty code cell and execute it by pressing *Shift+Enter*. This should execute the code, and create an new code cell. Pressing *Ctlr+Enter for Windows (or Cmd+Enter for Mac)* will execute the code without creating a new cell.

Commenting code in a code cell: Comments should be made while writing the code to explain the purpose of the code or a brief explanation of the tasks being performed by the code. A comment can be added in a code cell by preceding it with a # sign. For example, see the comment in the code below.

Writing comments will help other users understand your code. It is also useful for the coder to keep track of the tasks being performed by their code.

```
#This code adds 3 and 5
3+5
```

```
8
```

Markdown cell: Although a comment can be written in a code cell, a code cell cannot be used for writing headings/sub-headings, and is not appropriate for writing lengthy chunks of text. In such cases, change the cell type to *Markdown* from the dropdown menu below the *Widgets* tab. Use any markdown cheat sheet found online, for example, this one to format text in the markdown cells.

Give a name to te notebook by clicking on the text, which says 'Untitled'.

1.1.3 Saving and loading notebooks

Save the notebook by clicking on `File`, and selecting `Save as`, or clicking on the `Save and Checkpoint` icon (below the `File` tab). Your notebook will be saved as a file with an exptension *ipynb*. This file will contain all the code as well as the outputs, and can be loaded and edited by a Jupyter user. To load an existing Jupyter notebook, navigate to the folder of the notebook on the *landing page*, and then click on the file to open it.

1.1.4 Rendering notebook as HTML

We'll use Quarto to print the **.ipynb* file as HTML. Check the procedure for rendering a notebook as HTML here. You have several options to format the file.

You will need to open the command prompt, navigate to the directory containing the file, and use the command: `quarto render filename.ipynb --to html`.

In-class exercise

1. Create a new notebook.
2. Save the file as `In_class_exercise1`.
3. Give a heading to the file - `First HTML file`.
4. Print `Today is day 1 of class`.
5. Compute and print the number of hours of this course in the quarter (that will be 10 weeks x 2 classes per week x 1.33 hours per class).

The HTML file should look like the picture below.

```
<IPython.core.display.Image object>
```

1.2 Python language basics

1.2.1 Object Oriented Programming

Python is an object-oriented programming language. In layman terms, it means that every number, string, data structure, function, class, module, etc., exists in the python interpreter as a python object. An object may have attributes and methods associated with it. For example, let us define a variable that stores an integer:

```
var = 2
```

The variable `var` is an object that has attributes and methods associated with it. For example a couple of its attributes are `real` and `imag`, which store the real and imaginary parts respectively, of the object `var`:

```
print("Real part of 'var': ",var.real)
print("Real part of 'var': ",var.imag)
```

```
Real part of 'var':  2
Real part of 'var':  0
```

Attribute: An attribute is a value associated with an object, defined within the class of the object.

Method: A method is a function associated with an object, defined within the class of the object, and has access to the attributes associated with the object.

For looking at attributes and methods associated with an object, say `obj`, press tab key after typing `obj.`.

Consider the example below of a class *example_class*:

```
class example_class:
    class_name = 'My Class'
    def my_method(self):
        print('Hello World!')

e = example_class()
```

In the above class, `class_name` is an attribute, while `my_method` is a method.

1.2.2 Assigning variable name to object

When an object is assigned to a variable name, the variable name serves as a reference to the object. For example, consider the following assignment:

```
x = [5,3]
```

The variable name `x` is a reference to the memory location where the object `[5, 3]` is stored. Now, suppose we assign `x` to a new variable `y`:

```
y = x
```

In the above statement the variable name `y` now refers to the same object `[5,3]`. The object `[5,3]` does **not** get copied to a new memory location referred by `y`. To prove this, let us add an element to `y`:

```
y.append(4)
print(y)
```

```
[5, 3, 4]
```

```
print(x)
```

```
[5, 3, 4]
```

When we changed `y`, note that `x` also changed to the same object, showing that `x` and `y` refer to the same object, instead of referring to different copies of the same object.

1.2.3 Importing libraries

There are several built-in functions in python like `print()`, `abs()`, `max()`, `sum()` etc., which do not require importing any library. However, these functions will typically be insufficient for a analyzing data. Some of the popular libraries and their primary purposes are as follows:

1. NumPy: Performing numerical operations and efficiently storing numerical data.
2. Pandas: Reading, cleaning and manipulating data.
3. Matplotlib, Seaborn: Visualizing data.
4. SciPy: Performing scientific computing such as solving differential equations, optimization, statistical tests, etc.
5. Scikit-learn: Data pre-processing and machine learning, with a focus on prediction.
6. Statsmodels: Developing statistical models with a focus on inference

A library can be imported using the `import` keyword. For example, a NumPy library can be imported as:

```
import numpy as np
```

Using the `as` keyboard, the NumPy library has been given the name `np`. All the functions and attributes of the library can be called using the *'np.'* prefix. For example, let us generate a sequence of whole numbers upto `10` using the NumPy function arange():

```
np.arange(8)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

1.2.4 Built-in objects

There are several built-in objects, modules and functions in python. Below are a few examples:

Scalar objects: Python has some built-in datatypes for handling scalar objects such as number, string, boolean values, and date/time. The built-in function `type()` function can be used to determine the datatype of an object:

```
var = 2.2
type(var)
```

```
float
```

range(): The `range()` function returns a sequence of evenly-spaced integer values. It is commonly used in `for` loops to define the sequence of elements over which the iterations are performed.

Below is an example where the `range()` function is used to create a sequence of whole numbers upto 10:

```
print(list(range(1,10)))
```

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

Date time: Python as a built-in datetime module for handling date/time objects:

```
import datetime as dt
```

```
#Defining a date-time object
dt_object = dt.datetime(2022, 9, 20, 11,30,0)
```

Information about date and time can be accessed with the relevant attribute of the `datetime` object.

```
dt_object.day
```

```
20
```

```
dt_object.year
```

```
2022
```

The `strftime` method of the `datetime` module formats a `datetime` object as a string. There are several types of formats for representing date as a string:

```
dt_object.strftime('%m/%d/%Y')
```

```
'09/20/2022'
```

```
dt_object.strftime('%m/%d/%y %H:%M')
```

```
'09/20/22 11:30'
```

```
dt_object.strftime('%h-%d-%Y')
```

```
'Sep-20-2022'
```

1.2.5 Control flow

As in other languages, python has built-in keywords that provide conditional flow of control in the code.

If-elif-else: The `if-elif-else` statement can check several conditions, and execute the code corresponding to the condition that is true. Note that there can be as many `elif` statements as required.

```
#Example of if-elif-else
x = 5
if x>0:
    print("x is positive")
elif x==0:
    print("x is zero")
else:
    print("X is negative")
    print("This was the last condition checked")
```

```
x is positive
```

for loop: A `for` loop iterates over the elements of an object, and executes the statements within the loop in each iteration. For example, below is a `for` loop that prints odd natural numbers upto 10:

```
for i in range(10):
    if i%2!=0:
        print(i)
```

```
1
3
5
7
9
```

while loop: A `while` loop iterates over a set of statements *while* a condition is satisfied. For example, below is a `while` loop that prints odd numbers upto 10:

```
i=0
while i<10:
    if i%2!=0:
        print(i)
    i=i+1
```

```
1
3
5
7
9
```

2 Data structures

In this chapter we'll learn about the python data structures that are often used or appear while analyzing data.

Tuple is a sequence of python objects, with two key characeterisics: (1) the number of objects are fixed, and (2) the objects are immutable, i.e., they cannot be changed.

Tuple can be defined as a sequence of python objects separated by commas, and enclosed in rounded brackets (). For example, below is a tuple containing three integers.

```
tuple_example = (2,7,4)
```

We can check the data type of a python object using the *type()* function. Let us check the data type of the object *tuple_example*.

```
type(tuple_example)
```

```
tuple
```

Elements of a tuple can be extracted using their index within square brackets. For example the second element of the tuple *tuple_example* can be extracted as follows:

```
tuple_example[1]
```

```
7
```

Note that an element of a tuple cannot be modified. For example, consider the following attempt in changing the second element of the tuple *tuple_example*.

```
tuple_example[1] = 8
```

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

The above code results in an error as tuple elements cannot be modified.

2.0.1 Concatenating tuples

Tuples can be concatenated using the + operator to produce a longer tuple:

```
(2,7,4) + ("another", "tuple") + ("mixed","datatypes",5)
```

```
(2, 7, 4, 'another', 'tuple', 'mixed', 'datatypes', 5)
```

Multiplying a tuple by an integer results in repetition of the tuple:

```
(2,7,"hi") * 3
```

```
(2, 7, 'hi', 2, 7, 'hi', 2, 7, 'hi')
```

2.0.2 Unpacking tuples

If tuples are assigned to an expression containing multiple variables, the tuple will be unpacked and each variable will be assigned a value as per the order in which it appears. See the example below.

```
x,y,z  = (4.5, "this is a string", (("Nested tuple",5)))
```

```
x
```

```
4.5
```

```
y
```

```
'this is a string'
```

```
z
```

```
('Nested tuple', 5)
```

If we are interested in retrieving only some values of the tuple, the expression *_ can be used to discard the other values. Let's say we are interested in retrieving only the first and the last two values of the tuple:

```python
x,*_,y,z  = (4.5, "this is a string", (("Nested tuple",5)),"99",99)
```

```python
x
```

```
4.5
```

```python
y
```

```
'99'
```

```python
z
```

```
99
```

2.0.3 Tuple methods

A couple of useful tuple methods are `count`, which counts the occurences of an element in the tuple and `index`, which returns the position of the first occurance of an element in the tuple:

```python
tuple_example = (2,5,64,7,2,2)
```

```python
tuple_example.count(2)
```

```
3
```

```python
tuple_example.index(2)
```

```
0
```

Now that we have an idea about tuple, let us try to think where it can be used.

```
<IPython.core.display.HTML object>
```

2.1 List

List is a sequence of python objects, with two key characeterisics that differentiates it from tuple: (1) the number of objects are variable, i.e., objects can be added or removed from a list, and (2) the objects are mutable, i.e., they can be changed.

List can be defined as a sequence of python objects separated by commas, and enclosed in square brackets []. For example, below is a list consisting of three integers.

```
list_example = [2,7,4]
```

2.1.1 Adding and removing elements in a list

We can add elements at the end of the list using the *append* method. For example, we append the string 'red' to the list *list_example* below.

```
list_example.append('red')
```

```
list_example
```

```
[2, 7, 4, 'red']
```

Note that the objects of a list or a tuple can be of different datatypes.

An element can be added at a specific location of the list using the *insert* method. For example, if we wish to insert the number 2.32 as the second element of the list *list_example*, we can do it as follows:

```
list_example.insert(1,2.32)
```

```
list_example
```

```
[2, 2.32, 7, 4, 'red']
```

For removing an element from the list, the *pop* and *remove* methods may be used. The *pop* method removes an element at a particular index, while the *remove* method removes the element's first occurence in the list by its value. See the examples below.

Let us say, we need to remove the third element of the list.

```
list_example.pop(2)
```

```
7
```

```
list_example
```

```
[2, 2.32, 4, 'red']
```

Let us say, we need to remove the element 'red'.

```
list_example.remove('red')
```

```
list_example
```

```
[2, 2.32, 4]
```

```
#If there are multiple occurences of an element in the list, the first occurence will be r
list_example2 = [2,3,2,4,4]
list_example2.remove(2)
list_example2
```

```
[3, 2, 4, 4]
```

For removing multiple elements in a list, either `pop` or `remove` can be used in a `for` loop, or a `for` loop can be used with a condition. See the examples below.

Let's say we need to remove intergers less than 100 from the following list.

```
list_example3 = list(range(95,106))
list_example3
```

```
[95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105]
```

```
#Method 1: For loop with remove
list_example3_filtered = list(list_example3) #
```

```
for element in list_example3:
    if element<100:
        list_example3_filtered.remove(element)
print(list_example3_filtered)
```

```
[100, 101, 102, 103, 104, 105]
```

Q1: What's the need to define a new variable `list\_example3\_filtered` in the above code?

A1: Replace list_example3_filtered with list_example3 and identify the issue.

```
#Method 2: For loop with condition
[element for element in list_example3 if element>100]
```

```
[101, 102, 103, 104, 105]
```

2.1.2 List comprehensions

List comprehension is a compact way to create new lists based on elements of an existing list.

Example: Create a list that has squares of natural numbers from 5 to 15.

```
sqrt_natural_no_5_15 = [(x**2) for x in range(5,16)]
print(sqrt_natural_no_5_15)
```

```
[25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]
```

Example: Create a list of tuples, where each tuple consists of a natural number and its square, for natural numbers ranging from 5 to 15.

```
sqrt_natural_no_5_15 = [(x,x**2) for x in range(5,16)]
print(sqrt_natural_no_5_15)
```

```
[(5, 25), (6, 36), (7, 49), (8, 64), (9, 81), (10, 100), (11, 121), (12, 144), (13, 169), (1
```

2.1.3 Practice exercise 1

Below is a list consisting of responses to the question: "At what age do you think you will marry?" from students of the STAT303-1 Fall 2022 class.

```
exp_marriage_age=['24','30','28','29','30','27','26','28','30+','26','28','30','30','30','
```

Use list comprehension to:

2.1.3.1

Remove the elements that are not integers - such as *'probably never'*, *'30+'*, etc. What is the length of the new list?

Hint: The built-in python function of the `str` class - isdigit() may be useful to check if the string contains only digits.

Solution:

```
exp_marriage_age_num = [x for x in exp_marriage_age if x.isdigit()==True]
print("Length of the new list = ",len(exp_marriage_age_num))
```

```
Length of the new list =  181
```

2.1.3.2

Cap the values greater than 80 to 80, in the clean list obtained in (1). What is the mean age when people expect to marry in the new list?

```
exp_marriage_age_capped = [min(int(x),80) for x in exp_marriage_age_num]
print("Mean age when people expect to marry = ", sum(exp_marriage_age_capped)/len(exp_marr
```

```
Mean age when people expect to marry =  28.955801104972377
```

2.1.3.3

Determine the percentage of people who expect to marry at an age of 30 or more.

```
print("Percentage of people who expect to marry at an age of 30 or more =", str(100*sum([1
```

```
Percentage of people who expect to marry at an age of 30 or more = 37.01657458563536 %
```

2.1.4 Concatenating lists

As in tuples, lists can be concatenated using the `+` operator:

```
import time as tm
```

```
list_example4 = [5,'hi',4]
list_example4 = list_example4 + [None,'7',9]
list_example4
```

```
[5, 'hi', 4, None, '7', 9]
```

For adding elements to a list, the `extend` method is preferred over the `+` operator. This is because the `+` operator creates a new list, while the `extend` method adds elements to an existing list. Thus, the `extend` operator is more memory efficient.

```
list_example4 = [5,'hi',4]
list_example4.extend([None, '7', 9])
list_example4
```

```
[5, 'hi', 4, None, '7', 9]
```

2.1.5 Sorting a list

A list can be sorted using the `sort` method:

```
list_example5 = [6,78,9]
list_example5.sort(reverse=True) #the reverse argument is used to specify if the sorting i
list_example5
```

```
[78, 9, 6]
```

2.1.6 Slicing a list

We may extract or update a section of the list by passing the starting index (say `start`) and the stopping index (say `stop`) as `start:stop` to the index operator []. This is called *slicing* a list. For example, see the following example.

```
list_example6 = [4,7,3,5,7,1,5,87,5]
```

Let us extract a slice containing all the elements from the the 3rd position to the 7th position.

```
list_example6[2:7]
```

```
[3, 5, 7, 1, 5]
```

Note that while the element at the `start` index is included, the element with the `stop` index is excluded in the above slice.

If either the `start` or `stop` index is not mentioned, the slicing will be done from the beginning or until the end of the list, respectively.

```
list_example6[:7]
```

```
[4, 7, 3, 5, 7, 1, 5]
```

```
list_example6[2:]
```

```
[3, 5, 7, 1, 5, 87, 5]
```

To slice the list relative to the end, we can use negative indices:

```
list_example6[-4:]
```

```
[1, 5, 87, 5]
```

```
list_example6[-4:-2:]
```

```
[1, 5]
```

An extra colon (':') can be used to slice every nth element of a list.

```
#Selecting every 3rd element of a list
list_example6[::3]
```

```
[4, 5, 5]
```

```
#Selecting every 3rd element of a list from the end
list_example6[::-3]
```

```
[5, 1, 3]
```

```
#Selecting every element of a list from the end or reversing a list
list_example6[::-1]
```

```
[5, 87, 5, 1, 7, 5, 3, 7, 4]
```

2.1.7 Practice exercise 2

Start with the list [8,9,10]. Do the following:

2.1.7.1

Set the second entry (index 1) to 17

```
L = [8,9,10]
L[1]=17
```

2.1.7.2

Add 4, 5, and 6 to the end of the list

```
L = L+[4,5,6]
```

2.1.7.3

Remove the first entry from the list

```python
L.pop(0)
```

```
8
```

2.1.7.4

Sort the list

```python
L.sort()
```

2.1.7.5

Double the list (concatenate the list to itself)

```python
L=L+L
```

2.1.7.6

Insert 25 at index 3

The final list should equal [4,5,6,25,10,17,4,5,6,10,17]

```python
L.insert(3,25)
L
```

```
[4, 5, 6, 25, 10, 17, 4, 5, 6, 10, 17]
```

Now that we have an idea about lists, let us try to think where it can be used.

```
<IPython.core.display.HTML object>
```

Now that we have learned about lists and tuples, let us compare them.

Q2: A list seems to be much more flexible than tuple, and can replace a tuple almost everywhere. Then why use tuple at all?

A2: The additional flexibility of a list comes at the cost of efficiency. Some of the advatages of a tuple over a list are as follows:

1. Since a list can be extended, space is over-allocated when creating a list. A tuple takes less storage space as compared to a list of the same length.
2. Tuples are not copied. If a tuple is assigned to another tuple, both tuples point to the same memory location. However, if a list is assigned to another list, a new list is created consuming the same memory space as the orignial list.
3. Tuples refer to their element directly, while in a list, there is an extra layer of pointers that refers to their elements. Thus it is faster to retrieve elements from a tuple.

The examples below illustrate the above advantages of a tuple.

```
#Example showing tuples take less storage space than lists for the same elements
tuple_ex = (1, 2, 'Obama')
list_ex = [1, 2, 'Obama']
print("Space taken by tuple =",tuple_ex.__sizeof__()," bytes")
print("Space taken by list =",list_ex.__sizeof__()," bytes")
```

```
Space taken by tuple = 48  bytes
Space taken by list = 64  bytes
```

```
#Examples showing that a tuples are not copied, while lists can be copied
tuple_copy = tuple(tuple_ex)
print("Is tuple_copy same as tuple_ex?", tuple_ex is tuple_copy)
list_copy = list(list_ex)
print("Is list_copy same as list_ex?",list_ex is list_copy)
```

```
Is tuple_copy same as tuple_ex? True
Is list_copy same as list_ex? False
```

```
#Examples showing tuples takes lesser time to retrieve elements
import time as tm
tt = tm.time()
list_ex = list(range(1000000)) #List containinig whole numbers upto 1 million
a=(list_ex[::-2])
print("Time take to retrieve every 2nd element from a list = ", tm.time()-tt)

tt = tm.time()
tuple_ex = tuple(range(1000000)) #tuple containinig whole numbers upto 1 million
a=(tuple_ex[::-2])
print("Time take to retrieve every 2nd element from a tuple = ", tm.time()-tt)
```

```
Time take to retrieve every 2nd element from a list =  0.03579902648925781
Time take to retrieve every 2nd element from a tuple =  0.02684164047241211
```

2.2 Dictionary

A dictionary consists of key-value pairs, where the keys and values are python objects. While values can be any python object, keys need to be immutable python objects, like strings, intergers, tuples, etc. Thus, a list can be a value, but not a key, as elements of list can be changed. A dictionary is defined using the keyword `dict` along with curly braces, colons to separate keys and values, and commas to separate elements of a dictionary:

```
dict_example = {'USA':'Joe Biden', 'India':'Narendra Modi', 'China':'Xi Jinping'}
```

Elements of a dictionary can be retrieved by using the corresponding key.

```
dict_example['India']
```

```
'Narendra Modi'
```

2.2.1 Adding and removing elements in a dictionary

New elements can be added to a dictionary by defining a key in square brackets and assiging it to a value:

```
dict_example['Japan'] = 'Fumio Kishida'
dict_example['Countries'] = 4
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Japan': 'Fumio Kishida',
 'Countries': 4}
```

Elements can be removed from the dictionary using the `del` method or the `pop` method:

```
#Removing the element having key as 'Countries'
del dict_example['Countries']
```

```
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Japan': 'Fumio Kishida'}
```

```
#Removing the element having key as 'USA'
dict_example.pop('USA')
```

```
'Joe Biden'
```

```
dict_example
```

```
{'India': 'Narendra Modi', 'China': 'Xi Jinping', 'Japan': 'Fumio Kishida'}
```

New elements can be added, and values of exisiting keys can be changed using the `update` method:

```
dict_example = {'USA':'Joe Biden', 'India':'Narendra Modi', 'China':'Xi Jinping','Countrie
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Countries': 3}
```

```
dict_example.update({'Countries':4, 'Japan':'Fumio Kishida'})
```

```
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Countries': 4,
 'Japan': 'Fumio Kishida'}
```

2.2.2 Iterating over elements of a dictionary

The items() attribute of a dictionary can be used to iterate over elements of a dictionary.

```
for key,value in dict_example.items():
    print("The Head of State of",key,"is",value)
```

```
The Head of State of USA is Joe Biden
The Head of State of India is Narendra Modi
The Head of State of China is Xi Jinping
The Head of State of Countries is 4
The Head of State of Japan is Fumio Kishida
```

2.2.3 Practice exercise 3

The GDP per capita of USA for most years from 1960 to 2021 is given by the dictionary `D` given in the code cell below.

Find:

1. The GDP per capita in 2015
2. The GDP per capita of 2014 is missing. Update the dictionary to include the GDP per capita of 2014 as the average of the GDP per capita of 2013 and 2015.
3. Impute the GDP per capita of other missing years in the same manner as in (2), i.e., as the average GDP per capita of the previous year and the next year. Note that the GDP per capita is not missing for any two consecutive years.
4. Print the years and the imputed GDP per capita for the years having a missing value of GDP per capita in (3).

```
D = {'1960':3007,'1961':3067,'1962':3244,'1963':3375,'1964':3574,'1965':3828,'1966':4146,'
```

Solution:

```
print("GDP per capita in 2015 =", D['2015'])
D['2014'] = (D['2013']+D['2015'])/2
for i in range(1960,2021):
    if str(i) not in D.keys():
        D[str(i)] = (D[str(i-1)]+D[str(i+1)])/2
        print("Imputed GDP per capita for the year",i,"is $",D[str(i)])
```

```
GDP per capita in 2015 = 56763
Imputed GDP per capita for the year 1969 is $ 4965.0
Imputed GDP per capita for the year 1977 is $ 9578.5
Imputed GDP per capita for the year 1999 is $ 34592.0
```

2.3 Functions

If an algorithm or block of code is being used several times in a code, then it can be separately defined as a function. This makes the code more organized and readable. For example, let us define a function that prints prime numbers between `a` and `b`, and returns the number of prime numbers found.

```
#Function definition
def prime_numbers (a,b=100):
    num_prime_nos = 0

    #Iterating over all numbers between a and b
    for i in range(a,b):
        num_divisors=0

        #Checking if the ith number has any factors
        for j in range(2, i):
            if i%j == 0:
                num_divisors=1;break;

        #If there are no factors, then printing and counting the number as prime
        if num_divisors==0:
            print(i)
            num_prime_nos = num_prime_nos+1

    #Return count of the number of prime numbers
    return num_prime_nos
```

In the above function, the keyword `def` is used to define the function, `prime_numbers` is the name of the function, `a` and `b` are the arguments that the function uses to compute the output.

Let us use the defined function to print and count the prime numbers between 40 and 60.

```
#Printing prime numbers between 40 and 60
num_prime_nos_found = prime_numbers(40,60)
```

```
41
43
47
53
59
```

```
num_prime_nos_found
```

```
5
```

If the user calls the function without specifying the value of the argument `b`, then it will take the default value of `100`, as mentioned in the function definition. However, for the argument `a`, the user will need to specify a value, as there is no value defined as a default value in the function definition.

2.3.1 Global and local variables with respect to a function

A variable defined within a function is local to that function, while a variable defined outside the function is global to that function. In case a variable with the same name is defined both outside and inside a function, it will refer to its global value outside the function and local value within the function.

The example below shows a variable with the name `var` referring to its local value when called within the function, and global value when called outside the function.

```
var = 5
def sample_function(var):
    print("Local value of 'var' within 'sample_function()'= ",var)

sample_function(4)
print("Global value of 'var' outside 'sample_function()' = ",var)
```

```
Local value of 'var' within 'sample_function()'=  4
Global value of 'var' outside 'sample_function()' =  5
```

2.3.2 Practice exercise 4

The object `deck` defined below corresponds to a deck of cards. Estimate the probablity that a five card hand will be a flush, as follows:

1. Write a function that accepts a hand of 5 cards as argument, and returns whether the hand is a flush or not.
2. Randomly pull a hand of 5 cards from the deck. Call the function developed in (1) to determine if the hand is a flush.
3. Repeat (2) 10,000 times.
4. Estimate the probability of the hand being a flush from the results of the 10,000 simulations.

You may use the function shuffle() from the `random` library to shuffle the deck everytime before pulling a hand of 5 cards.

```
deck = [{'value':i, 'suit':c}
for c in ['spades', 'clubs', 'hearts', 'diamonds']
for i in range(2,15)]
```

Solution:

```
import random as rm

#Function to check if a 5-card hand is a flush
def chck_flush(hands):

    #Assuming that the hand is a flush, before checking the cards
    yes_flush =1

    #Storing the suit of the first card in 'first_suit'
    first_suit = hands[0]['suit']

    #Iterating over the remaining 4 cards of the hand
    for j in range(1,len(hands)):

        #If the suit of any of the cards does not match the suit of the first card, the ha
        if first_suit!=hands[j]['suit']:
            yes_flush = 0;

            #As soon as a card with a different suit is found, the hand is not a flush and
            break;
    return yes_flush

flush=0
for i in range(10000):
```

```python
    #Shuffling the deck
    rm.shuffle(deck)

    #Picking out the first 5 cards of the deck as a hand and checking if they are a flush
    #If the hand is a flush it is counted
    flush=flush+chck_flush(deck[0:5])

print("Probability of obtaining a flush=", 100*(flush/10000),"%")
```

```
Probability of obtaining a flush= 0.18 %
```

2.4 Practice exercise 5

The code cell below defines an object having the nutrition information of drinks in starbucks. Assume that the manner in which the information is structured is consistent throughout the object.

```python
starbucks_drinks_nutrition={'Cool Lime Starbucks Refreshers Beverage': [{'Nutrition_type': 'Cal
, 'value': 1}, {'Nutrition_type': 'Sodium', 'value': 10}], 'Featured Dark Roast': [{'Nutrition_type': 'Calo
```

Use the object above to answer the following questions:

2.4.1

What is the datatype of the object?

```python
print("Datatype=",type(starbucks_drinks_nutrition))
```

```
Datatype= <class 'dict'>
```

2.4.1.1

If the object in (1) is a dictonary, what is the datatype of the values of the dictionary?

```python
print("Datatype=",type(starbucks_drinks_nutrition[list(starbucks_drinks_nutrition.keys())[
```

```
Datatype= <class 'list'>
```

2.4.1.2

If the object in (1) is a dictonary, what is the datatype of the elements within the values of the dictionary?

```
print("Datatype=",type(starbucks_drinks_nutrition[list(starbucks_drinks_nutrition.keys())[
```

```
Datatype= <class 'dict'>
```

2.4.1.3

How many calories are there in `Iced Coffee`?

```
print("Calories = ",starbucks_drinks_nutrition['Iced Coffee'][0]['value'])
```

```
Calories =  5
```

2.4.1.4

Which drink(s) have the highest amount of protein in them, and what is that protein amount?

```
#Defining an empty dictionary that will be used to store the protein of each drink
protein={}

for key,value in starbucks_drinks_nutrition.items():
    for nutrition in value:
        if nutrition['Nutrition_type']=='Protein':
            protein[key]=(nutrition['value'])

#Using dictionary comprehension to find the key-value pair having the maximum value in the
{key:value for key, value in protein.items() if value == max(protein.values())}
```

```
{'Starbucks® Doubleshot Protein Dark Chocolate': 20,
 'Starbucks® Doubleshot Protein Vanilla': 20,
 'Chocolate Smoothie': 20}
```

2.4.1.5

Which drink(s) have a fat content of more than 10g, and what is their fat content?

```
#Defining an empty dictionary that will be used to store the fat of each drink
fat={}
for key,value in starbucks_drinks_nutrition.items():
    for nutrition in value:
        if nutrition['Nutrition_type']=='Fat':
            fat[key]=(nutrition['value'])

#Using dictionary comprehension to find the key-value pair having the value more than 10
{key:value for key, value in fat.items() if value>=10}
```

```
{'Starbucks® Signature Hot Chocolate': 26.0, 'White Chocolate Mocha': 11.0}
```

3 Reading data

Reading data is the first step to extract information from it. Data can exist broadly in two formats:

(1) Structured data, and
(2) Untructured data.

Structured data is typically stored in a tabular form, where rows in the data correspond to "observations" and columns correspond to "variables". For example, the following dataset contains 5 observations, where each observation (or row) consists of information about a movie. The variables (or columns) contain different pieces of information about a given movie. As all variables for a given row are related to the same movie, the data below is also called relational data.

	Title	US Gross	Production Budget	Release Date	Major Genre
0	The Shawshank Redemption	28241469	25000000	Sep 23 1994	Drama
1	Inception	285630280	160000000	Jul 16 2010	Horror/Thriller
2	One Flew Over the Cuckoo's Nest	108981275	4400000	Nov 19 1975	Comedy
3	The Dark Knight	533345358	185000000	Jul 18 2008	Action/Adventure
4	Schindler's List	96067179	25000000	Dec 15 1993	Drama

Unstructured data is data that is not organized in any pre-defined manner. Examples of unstructured data can be text files, audio/video files, images, Internet of Things (IoT) data, etc. Unstructured data is relatively harder to analyze as most of the analytical methods and tools are oriented towards structured data. However, an unstructured data can be used to obtain structured data, which in turn can be analyzed. For example, an image can be converted to an array of pixels - which will be structured data. Machine learning algorithms can then be used on the array to classify the image as that of a dog or a cat.

In this course, we will focus on analyzing structured data.

3.1 Reading a *csv* file with *Pandas*

Structured data can be stored in a variety of formats. The most popular format is *data_file_name.csv*, where the extension *csv* stands for comma separated values. The variable values of each observation are separated by a comma in a *.csv* file. In other words, the **delimiter** is a comma in a *csv* file. However, the comma is not visible when a *.csv* file is opened with Microsoft Excel.

3.1.1 Using the *read_csv* function

We will use functions from the *Pandas* library of *Python* to read data. Let us import *Pandas* to use its functions.

```
import pandas as pd
```

Note that *pd* is the acronym that we will use to call a *Pandas* function. This acronym can be anything as desired by the user.

The function to read a *csv* file is read_csv(). It reads the dataset into an object of type Pandas DataFrame. Let us read the dataset *movie_ratings.csv* in Python.

```
movie_ratings = pd.read_csv('movie_ratings.csv')
```

The built-in python function `type` can be used to check the dataype of an object:

```
type(movie_ratings)
```

```
pandas.core.frame.DataFrame
```

Note that the file *movie_ratings.csv* is stored at the same location as the python script containing the above code. If that is not the case, we'll need to specify the location of the file as in the following code.

```
movie_ratings = pd.read_csv('D:/Books/DataScience_Intro_python/movie_ratings.csv')
```

Note that forward slash is used instead of backslash while specifying the path of the data file. Another option is to use two consecutive backslashes instead of a single forward slash.

3.1.2 Specifying the working directory

In case we need to read several datasets from a given location, it may be inconvenient to specify the path every time. In such a case we can change the current working directory to the location where the datasets are located.

We'll use the *os* library of *Python* to view and/or change the current working directory.

```
import os #Importing the 'os' library
os.getcwd() #Getting the path to the current working directory
```

```
C:\Users\username\STAT303-1\Quarto Book\DataScience_Intro_python
```

The function *getcwd()* stands for get current working directory.

Suppose the dataset to be read is located at `'D:\Books\DataScience_Intro_python\Datasets'`. Then, we'll use the function *chdir* to change the current working directory to this location.

```
os.chdir('D:/Books/DataScience_Intro_python/Datasets')
```

Now we can read the dataset from this location without mentioning the entire path as shown below.

```
movie_ratings = pd.read_csv('movie_ratings.csv')
```

3.1.3 Data overview

Once the data has been read, we may want to see what the data looks like. We'll use another *Pandas* function head() to view the first few rows of the data.

```
movie_ratings.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
0	Opal Dreams	14443	14443	9000000	Nov 22 2006	PG/PG-13
1	Major Dundee	14873	14873	3800000	Apr 07 1965	PG/PG-13
2	The Informers	315000	315000	18000000	Apr 24 2009	R
3	Buffalo Soldiers	353743	353743	15000000	Jul 25 2003	R
4	The Last Sin Eater	388390	388390	2200000	Feb 09 2007	PG/PG-13

3.1.3.1 Row Indices and column names (axis labels)

The bold integers on the left are the indices of the DataFrame. Each index refers to a distinct row. For example, the index *2* correponds to the row of the movie *The Informers*. By default, the indices are integers starting from 0. However, they can be changed (to even non-integer values) if desired by the user.

The bold text on top of the DataFrame refers to column names. For example, the column *US Gross* consists of the gross revenue of a movie in the US.

Collectively, the indices and column names are referred as **axis labels**.

3.1.3.2 Shape of DataFrame

For finding the number of rows and columns in the data, you may use the shape() function.

```
#Finding the shape of movie_ratings dataset
movie_ratings.shape
```

```
(2228, 11)
```

The *movie_ratings* dataset contains 2,228 observations (or rows) and 11 variables (or columns).

3.1.4 Summary statistics

3.1.4.1 Numeric columns summary

The Pandas function of the DataFrame class, describe() can be used very conviniently to print the summary statistics of numeric columns of the data.

```
#Finding summary statistics of movie_ratings dataset
movie_ratings.describe()
```

Table 3.3: Summary statistics of numeric variables

	US Gross	Worldwide Gross	Production Budget	IMDB Rating	IMDB Votes	Release Yea
count	2.228000e+03	2.228000e+03	2.228000e+03	2228.000000	2228.000000	2228.000000
mean	5.076370e+07	1.019370e+08	3.816055e+07	6.239004	33585.154847	2002.005386
std	6.643081e+07	1.648589e+08	3.782604e+07	1.243285	47325.651561	5.524324
min	0.000000e+00	8.840000e+02	2.180000e+02	1.400000	18.000000	1953.000000
25%	9.646188e+06	1.320737e+07	1.200000e+07	5.500000	6659.250000	1999.000000
50%	2.838649e+07	4.266892e+07	2.600000e+07	6.400000	18169.000000	2002.000000
75%	6.453140e+07	1.200000e+08	5.300000e+07	7.100000	40092.750000	2006.000000
max	7.601676e+08	2.767891e+09	3.000000e+08	9.200000	519541.000000	2039.000000

Answer the following questions based on the above table.

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

3.1.4.2 Summary statistics across rows/columns

The Pandas DataFrame class has functions such as sum() and mean() to compute sum over rows or columns of a DataFrame.

Let us compute the mean of all the numeric columns of the data:

```
movie_ratings.mean(axis = 0)
```

```
US Gross            5.076370e+07
Worldwide Gross     1.019370e+08
Production Budget   3.816055e+07
IMDB Rating         6.239004e+00
IMDB Votes          3.358515e+04
dtype: float64
```

The argument `axis=0` denotes that the mean is taken over all the rows of the DataFrame. For computing a statistic across column the argument `axis=1` will be used.

If mean over a subset of columns is desired, then those column names can be subset from the data. For example, let us compute the mean IMDB rating, and mean IMDB votes of all the movies:

```
movie_ratings[['IMDB Rating','IMDB Votes']].mean(axis = 0)
```

```
IMDB Rating        6.239004
IMDB Votes     33585.154847
dtype: float64
```

3.1.5 Practice exercise 1

Read the file *Top 10 Albums By Year.csv*. This file contains the top 10 albums for each year from 1990 to 2021. Each row corresponds to a unique album.

3.1.5.1

Print the first 5 rows of the data.

```
album_data = pd.read_csv('./Datasets/Top 10 Albums By Year.csv')
album_data.head()
```

	Year	Ranking	Artist	Album	Worldwide Sales	Cl
0	1990	8	Phil Collins	Serious Hits... Live!	9956520	1
1	1990	1	Madonna	The Immaculate Collection	30000000	1
2	1990	10	The Three Tenors	Carreras Domingo Pavarotti In Concert 1990	8533000	1
3	1990	4	MC Hammer	Please Hammer Don't Hurt Em	18000000	1
4	1990	6	Movie Soundtrack	Aashiqui	15000000	1

3.1.5.2

How many rows and columns are there in the data?

```
album_data.shape
```

```
(320, 12)
```

There are 320 rows and 12 columns in the data

3.1.5.3

Print the summary statistics of the data, and answer the following questions:

1. What proportion of albums have 15 or lesser tracks? Mention a range for the proportion.
2. What is the mean length of a track (in minutes)?

```
album_data.describe()
```

	Year	Ranking	CDs	Tracks	Hours	Minutes	Seconds
count	320.000000	320.00000	320.000000	320.000000	320.000000	320.000000	320.000000
mean	2005.500000	5.50000	1.043750	14.306250	0.941406	56.478500	3388.715625
std	9.247553	2.87678	0.246528	5.868995	0.382895	22.970109	1378.209812
min	1990.000000	1.00000	1.000000	6.000000	0.320000	19.430000	1166.000000

	Year	Ranking	CDs	Tracks	Hours	Minutes	Seconds
25%	1997.750000	3.00000	1.000000	12.000000	0.740000	44.137500	2648.250000
50%	2005.500000	5.50000	1.000000	13.000000	0.860000	51.555000	3093.500000
75%	2013.250000	8.00000	1.000000	15.000000	1.090000	65.112500	3906.750000
max	2021.000000	10.00000	4.000000	67.000000	5.070000	304.030000	18242.000000

At least 75% of the albums have 15 tracks since the 75th percentile value of the number of tracks is 15. However, albums between those having 75th percentile value for the number of tracks and those having the maximum number of tracks can also have 15 tracks. Thus, the proportion of albums having 15 or lesser tracks = [75%-99.99%].

```
print("Mean length of a track =",56.478500/14.306250, "minutes")
```

```
Mean length of a track = 3.9478200087374398 minutes
```

3.1.6 Creating new columns from existing columns

New variables (or columns) can be created based on existing variables, or with external data (we'll see adding external data later). For example, let us create a new variable `ratio_wgross_by_budget`, which is the ratio of `Worldwide Gross` and `Production Budget` for each movie:

```
movie_ratings['ratio_wgross_by_budget'] = movie_ratings['Worldwide Gross']/movie_ratings['
```

The new variable can be seen at the right end of the updated DataFrame as shown below.

```
movie_ratings.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
0	Opal Dreams	14443	14443	9000000	Nov 22 2006	PG/PG-13
1	Major Dundee	14873	14873	3800000	Apr 07 1965	PG/PG-13
2	The Informers	315000	315000	18000000	Apr 24 2009	R
3	Buffalo Soldiers	353743	353743	15000000	Jul 25 2003	R
4	The Last Sin Eater	388390	388390	2200000	Feb 09 2007	PG/PG-13

3.1.7 Datatype of variables

Note that in Table 3.3 (summary statistics), we don't see `Release Date`. This is because the datatype of `Release Data` is not `numeric`.

The datatype of each variable can be seen using the dtypes() function of the DataFrame class.

```
#Checking the datatypes of the variables
movie_ratings.dtypes
```

```
Title                   object
US Gross                 int64
Worldwide Gross          int64
Production Budget        int64
Release Date            object
MPAA Rating             object
Source                  object
Major Genre             object
Creative Type           object
IMDB Rating            float64
IMDB Votes               int64
dtype: object
```

Often, we wish to convert the datatypes of some of the variables to make them suitable for analysis. For example, the datatype of `Release Date` in the DataFrame *movie_ratings* is `object`. To perform numerical computations on this variable, we'll need to convert it to a `datatime` format. We'll use the Pandas function to_datatime() to covert it to a `datatime` format. Similar functions such as to_numeric(), to_string() etc., can be used for other conversions.

```
pd.to_datetime(movie_ratings['Release Date'])
```

```
0       2006-11-22
1       1965-04-07
2       2009-04-24
3       2003-07-25
4       2007-02-09
           ...
2223    2004-07-07
2224    1998-06-19
2225    2010-05-14
```

```
2226   1991-06-14
2227   1998-01-23
Name: Release Date, Length: 2228, dtype: datetime64[ns]
```

We can see above that the function `to_datetime()` converts *Release Date* to a `datetime` format.

Now, we'll update the variable `Release Date` in the DataFrame to be in the `datetime` format:

```
movie_ratings['Release Date'] = pd.to_datetime(movie_ratings['Release Date'])
```

```
movie_ratings.dtypes
```

```
Title                        object
US Gross                      int64
Worldwide Gross               int64
Production Budget             int64
Release Date         datetime64[ns]
MPAA Rating                  object
Source                       object
Major Genre                  object
Creative Type                object
IMDB Rating                 float64
IMDB Votes                    int64
dtype: object
```

We can see that the datatype of *Release Date* has changed to `datetime` in the updated DataFrame, *movie_ratings*. Now we can perform computations on `Release Date`. Suppose we wish to create a new variable `Release_year` that consists of the year of release of the movie. We'll use the attribute `year` of the datetime module to extract the year from `Release Date`:

```
#Extracting year from Release Date
movie_ratings['Release Year'] = movie_ratings['Release Date'].dt.year
```

```
movie_ratings.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
0	Opal Dreams	14443	14443	9000000	2006-11-22	PG/PG-13

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MPAA Rating
1	Major Dundee	14873	14873	3800000	1965-04-07	PG/PG-13
2	The Informers	315000	315000	18000000	2009-04-24	R
3	Buffalo Soldiers	353743	353743	15000000	2003-07-25	R
4	The Last Sin Eater	388390	388390	2200000	2007-02-09	PG/PG-13

As year is a numeric variable, it will appear in the numeric summary statistics with the `describe()` function, as shown below.

```
movie_ratings.describe()
```

	US Gross	Worldwide Gross	Production Budget	IMDB Rating	IMDB Votes	Release Yea
count	2.228000e+03	2.228000e+03	2.228000e+03	2228.000000	2228.000000	2228.000000
mean	5.076370e+07	1.019370e+08	3.816055e+07	6.239004	33585.154847	2002.005386
std	6.643081e+07	1.648589e+08	3.782604e+07	1.243285	47325.651561	5.524324
min	0.000000e+00	8.840000e+02	2.180000e+02	1.400000	18.000000	1953.000000
25%	9.646188e+06	1.320737e+07	1.200000e+07	5.500000	6659.250000	1999.000000
50%	2.838649e+07	4.266892e+07	2.600000e+07	6.400000	18169.000000	2002.000000
75%	6.453140e+07	1.200000e+08	5.300000e+07	7.100000	40092.750000	2006.000000
max	7.601676e+08	2.767891e+09	3.000000e+08	9.200000	519541.000000	2039.000000

3.1.8 Practice exercise 2

3.1.8.1

Why is `Worldwide Sales` not included in the summary statistics table printed in Practice exercise 1?

```
album_data.dtypes
```

```
Year                  int64
Ranking               int64
Artist               object
Album                object
Worldwide Sales      object
CDs                   int64
Tracks                int64
Album Length         object
```

```
Hours               float64
Minutes             float64
Seconds               int64
Genre                object
dtype: object
```

`Worldwide Sales` is not included in the summary statistics table printed in Practice exercise 1 because its data type is `object` and not `int` or `float`

3.1.8.2

Update the DataFrame so that `Worldwide Sales` is included in the summary statistics table. Print the summary statistics table.

Hint: Sometimes it may not be possible to convert an object to numeric(). For example, the object 'hi' cannot be converted to a numeric() by the python compiler. To avoid getting an error, use the *errors* argument of to_numeric() to force such conversions to NaN (missing value).

```
album_data['Worldwide Sales'] = pd.to_numeric(album_data['Worldwide Sales'], errors = 'coe
album_data.describe()
```

	Year	Ranking	Worldwide Sales	CDs	Tracks	Hours	Minutes	Sec
count	320.000000	320.00000	3.190000e+02	320.000000	320.000000	320.000000	320.000000	320
mean	2005.500000	5.50000	1.071093e+07	1.043750	14.306250	0.941406	56.478500	338
std	9.247553	2.87678	7.566796e+06	0.246528	5.868995	0.382895	22.970109	137
min	1990.000000	1.00000	1.909009e+06	1.000000	6.000000	0.320000	19.430000	116
25%	1997.750000	3.00000	5.000000e+06	1.000000	12.000000	0.740000	44.137500	264
50%	2005.500000	5.50000	8.255866e+06	1.000000	13.000000	0.860000	51.555000	309
75%	2013.250000	8.00000	1.400000e+07	1.000000	15.000000	1.090000	65.112500	390
max	2021.000000	10.00000	4.500000e+07	4.000000	67.000000	5.070000	304.030000	182

3.1.8.3

Create a new column that computes the average worldwide sales per year for each album, assuming that the worldwide sales are as of 2022. Print the first 5 rows of the updated DataFrame.

```
album_data['mean_sales_per_year'] = album_data['Worldwide Sales']/(2022-album_data['Year']
album_data.head()
```

	Year	Ranking	Artist	Album	Worldwide Sales	Cl
0	1990	8	Phil Collins	Serious Hits... Live!	9956520.0	1
1	1990	1	Madonna	The Immaculate Collection	30000000.0	1
2	1990	10	The Three Tenors	Carreras Domingo Pavarotti In Concert 1990	8533000.0	1
3	1990	4	MC Hammer	Please Hammer Don't Hurt Em	18000000.0	1
4	1990	6	Movie Soundtrack	Aashiqui	15000000.0	1

3.1.9 Reading a sub-set of data: `loc` and `iloc`

Sometimes we may be interested in working with a subset of rows and columns of the data, instead of working with the entire dataset. The indexing operators loc and iloc provide a convenient way of selecting a subset of desired rows and columns. The operator `loc` uses axis labels (row indices and column names) to subset the data, while `iloc` uses the position of rows or columns, where position has values 0,1,2,3,…and so on, for rows from top to bottom and columns from left to right. In other words, the first row has position 0, the second row has position 1, the third row has position 2, and so on. Similarly, the first column from left has position 0, the second column from left has position 1, the third column from left has position 2, and so on.

Let us read the file *movie_IMDBratings_sorted.csv*, which has movies sorted in the descending order of their IMDB ratings.

```
movies_sorted = pd.read_csv('./Datasets/movie_IMDBratings_sorted.csv',index_col = 0)
```

The argument `index_col=0` assigns the first column of the file as the row indices of the DataFrame.

```
movies_sorted.head()
```

	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MI
Rank						
1	The Shawshank Redemption	28241469	28241469	25000000	Sep 23 1994	R
2	Inception	285630280	753830280	160000000	Jul 16 2010	PG
3	The Dark Knight	533345358	1022345358	185000000	Jul 18 2008	PG
4	Schindler's List	96067179	321200000	25000000	Dec 15 1993	R
5	Pulp Fiction	107928762	212928762	8000000	Oct 14 1994	R

Let us say, we wish to subset the title, worldwide gross, production budget, and IMDB raring of top 3 movies.

```
# Subsetting the DataFrame by loc - using axis labels
movies_subset = movies_sorted.loc[1:3,['Title','Worldwide Gross','Production Budget','IMDB
movies_subset
```

Rank	Title	Worldwide Gross	Production Budget	IMDB Rating
1	The Shawshank Redemption	28241469	25000000	9.2
2	Inception	753830280	160000000	9.1
3	The Dark Knight	1022345358	185000000	8.9

```
# Subsetting the DataFrame by iloc - using index of the position of rows and columns
movies_subset = movies_sorted.iloc[0:3,[0,2,3,9]]
movies_subset
```

Rank	Title	Worldwide Gross	Production Budget	IMDB Rating
1	The Shawshank Redemption	28241469	25000000	9.2
2	Inception	753830280	160000000	9.1
3	The Dark Knight	1022345358	185000000	8.9

Let us find the movie with the maximum `Worldwide Gross`.

We will use the argmax() function of the Pandas Series class to find the position of the movie with the maximum worldwide gross, and then use the position to find the movie.

```
position_max_wgross = movies_sorted['Worldwide Gross'].argmax()
```

```
movies_sorted.iloc[position_max_wgross,:]
```

```
Title                             Avatar
US Gross                       760167650
Worldwide Gross               2767891499
Production Budget              237000000
Release Date                 Dec 18 2009
MPAA Rating                    PG/PG-13
Source               Original Screenplay
Major Genre             Action/Adventure
Creative Type                    Fiction
```

```
IMDB Rating                         8.3
IMDB Votes                       261439
Name: 59, dtype: object
```

Avatar has the highest worldwide gross of all the movies. Note that the `:` indicates that all the columns of the DataFrame are selected.

3.1.10 Practice exercise 3

3.1.10.1

Find the album having the highest worldwide sales per year, and its artist.

```
album_data.iloc[album_data['mean_sales_per_year'].argmax(),:]
```

```
Year                        2021
Ranking                        1
Artist                     Adele
Album                         30
Worldwide Sales        4485025.0
CDs                            1
Tracks                        12
Album Length             0:58:14
Hours                       0.97
Minutes                    58.23
Seconds                     3494
Genre                        Pop
mean_sales_per_year    4485025.0
Name: 312, dtype: object
```

'30' has the highest worldwide sales and its artist is Adele.

3.1.10.2

Subset the data to include only Hip-Hop albums. How many Hip_Hop albums are there?

```
hiphop_albums = album_data.loc[album_data['Genre']=='Hip Hop',:]
print("There are",hiphop_albums.shape[0], "hip-hop albums")
```

```
There are 42 hip-hop albums
```

3.1.10.3

Which album amongst hip-hop has the higest mean sales per year per track, and who is its artist?

```
hiphop_albums.loc[:,'mean_sales_per_year_track'] = hiphop_albums.loc[:,'Worldwide Sales']/
hiphop_albums.iloc[hiphop_albums['mean_sales_per_year_track'].argmax(),:]
```

```
Year                                2021
Ranking                                6
Artist                         Cai Xukun
Album
Worldwide Sales                3402981.0
CDs                                    1
Tracks                                11
Album Length                     0:24:16
Hours                                0.4
Minutes                            24.27
Seconds                             1456
Genre                            Hip Hop
mean_sales_per_year            3402981.0
mean_sales_per_year_track  309361.909091
Name: 318, dtype: object
```

has the higest mean sales per year per track amongst hip-hop albumns, and its artist is Cai Xukun.

3.2 Reading other data formats - txt, html, json

Although *csv* is a very popular format for structured data, data is found in several other formats as well. Some of the other data formats are *txt, html* and *json*.

3.2.1 Reading *txt* files

The *txt* format offers some additional flexibility as compared to the *csv* format. In the *csv* format, the delimiter is a comma (or the column values are separated by a comma). However, in a *txt* file, the delimiter can be anything as desired by the user. Let us read the file *movie_ratings.txt*, where the variable values are separated by a tab character.

```
movie_ratings_txt = pd.read_csv('movie_ratings.txt',sep='\t')
```

We use the function *read_csv* to read a *txt* file. However, we mention the tab character (r"") as a separator of variable values.

Note that there is no need to remember the argument name - *sep* for specifying the delimiter. You can always refer to the read_csv() documentation to find the relevant argument.

3.2.2 Practice exercise 4

Read the file *bestseller_books.txt*. It contains top 50 best-selling books on amazon from 2009 to 2019. Identify the delimiter without opening the file with Notepad or a text-editing software. How many rows and columns are there in the dataset?

Solution:

```
#Reading some lines with 'error_bad_lines=False' to identify the delimiter
bestseller_books = pd.read_csv('./Datasets/bestseller_books.txt',error_bad_lines=False)
bestseller_books.head()
```

```
b'Skipping line 6: expected 1 fields, saw 2\nSkipping line 10: expected 1 fields, saw 3\nSki
```

	;Unnamed: 0;Name;Author;User Rating;Reviews;Price;Year;Genre
0	0;0;10-Day Green Smoothie Cleanse;JJ Smith;4.7...
1	1;1;11/22/63: A Novel;Stephen King;4.6;2052;22...
2	2;2;12 Rules for Life: An Antidote to Chaos;Jo...
3	3;3;1984 (Signet Classics);George Orwell;4.7;2...
4	5;5;A Dance with Dragons (A Song of Ice and Fi...

```
#The delimiter seems to be ';' based on the output of the above code
bestseller_books = pd.read_csv('./Datasets/bestseller_books.txt',sep=';')
bestseller_books.head()
```

	Unnamed: 0	Unnamed: 0.1	Name	Author
0	0	0	10-Day Green Smoothie Cleanse	JJ Smith
1	1	1	11/22/63: A Novel	Stephen King
2	2	2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson
3	3	3	1984 (Signet Classics)	George Orwell

	Unnamed: 0	Unnamed: 0.1	Name	Author
4	4	4	5,000 Awesome Facts (About Everything!) (Natio...	National Geographic

```
#The file read with ';' as the delimited is correct
print("The file has",bestseller_books.shape[0],"rows and",bestseller_books.shape[1],"colum
```

```
The file has 550 rows and 9 columns
```

Alternatively, you can use the argument `sep = None`, and `engine = 'python'`. The default engine is C. However, the 'python' engine has a 'sniffer' tool which may identify the delimiter automatically.

```
bestseller_books = pd.read_csv('./Datasets/bestseller_books.txt',sep=None, engine = 'pytho
bestseller_books.head()
```

	Unnamed: 0	Unnamed: 0.1	Name	Author
0	0	0	10-Day Green Smoothie Cleanse	JJ Smith
1	1	1	11/22/63: A Novel	Stephen King
2	2	2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson
3	3	3	1984 (Signet Classics)	George Orwell
4	4	4	5,000 Awesome Facts (About Everything!) (Natio...	National Geographic

3.2.3 Reading HTML data

The *Pandas* function *read_html* searches for tabular data, i.e., data contained within the *<table>* tags of an html file. Let us read the tables in the GDP per capita page on Wikipedia.

```
#Reading all the tables from the Wikipedia page on GDP per capita
tables = pd.read_html('https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)_pe
```

All the tables will be read and stored in the variable named as *tables*. Let us find the datatype of the variable *tables*.

```
#Finidng datatype of the variable - tables
type(tables)
```

```
list
```

The variable - tables is a list of all the tables read from the HTML data.

```
#Number of tables read from the page
len(tables)
```

```
6
```

The in-built function *len* can be used to find the length of the list - *tables* or the number of tables read from the Wikipedia page. Let us check out the first table.

```
#Checking out the first table. Note that the index of the first table will be 0.
tables[0]
```

	0	1	2
0	.mw-parser-output .legend{page-break-inside:av...	$20,000 - $30,000 $10,000 - $20,000 $5,000 - $...	$1,00

The above table doesn't seem to be useful. Let us check out the second table.

```
#Checking out the second table. Note that the index of the first table will be 1.
tables[1]
```

	Country/Territory	UN Region	IMF[4][5]		United Nations[6]		World Bank[7]	
	Country/Territory	UN Region	Estimate	Year	Estimate	Year	Estimate	Year
0	Liechtenstein *	Europe	—	—	180227	2020	169049	2019
1	Monaco *	Europe	—	—	173696	2020	173688	2020
2	Luxembourg *	Europe	135046	2022	117182	2020	135683	2021
3	Bermuda *	Americas	—	—	123945	2020	110870	2021
4	Ireland *	Europe	101509	2022	86251	2020	85268	2020
...	...	...	...	...	...	...	...	...
212	Central AfricanRepublic *	Africa	527	2022	481	2020	477	2020
213	Sierra Leone *	Africa	513	2022	475	2020	485	2020
214	Madagascar *	Africa	504	2022	470	2020	496	2020
215	South Sudan *	Africa	393	2022	1421	2020	1120	2015
216	Burundi *	Africa	272	2022	286	2020	274	2020

The above table contains the estimated GDP per capita of all countries. This is the table that is likely to be relevant to a user interested in analyzing GDP per capita of countries. Instead of reading all tables of an HTML file, we can focus the search to tables containing certain relevant keywords. Let us try searching all table containing the word 'Country'.

```
#Reading all the tables from the Wikipedia page on GDP per capita, containing the word 'Co
tables = pd.read_html('https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)_pe
```

The *match* argument can be used to specify the kewyords to be present in the table to be read.

```
len(tables)
```

```
1
```

Only one table contains the keyword - 'Country'. Let us check out the table obtained.

```
#Table having the keyword - 'Country' from the HTML page
tables[0]
```

	Country/Territory	UN Region	IMF[4][5]		United Nations[6]		World Bank[7]	
	Country/Territory	UN Region	Estimate	Year	Estimate	Year	Estimate	Year
0	Liechtenstein *	Europe	—	—	180227	2020	169049	2019
1	Monaco *	Europe	—	—	173696	2020	173688	2020
2	Luxembourg *	Europe	135046	2022	117182	2020	135683	2021
3	Bermuda *	Americas	—	—	123945	2020	110870	2021
4	Ireland *	Europe	101509	2022	86251	2020	85268	2020
...	...	...	...	...	...	...	...	...
212	Central AfricanRepublic *	Africa	527	2022	481	2020	477	2020
213	Sierra Leone *	Africa	513	2022	475	2020	485	2020
214	Madagascar *	Africa	504	2022	470	2020	496	2020
215	South Sudan *	Africa	393	2022	1421	2020	1120	2015
216	Burundi *	Africa	272	2022	286	2020	274	2020

The argument *match* helps with a more focussed search, and helps us discard irrelevant tables.

3.2.4 Practice exercise 5

Read the table(s) consisting of attendance of spectators in FIFA worlds cup from this page. Read only those table(s) that have the word *'attendance'* in them. How many rows and columns are there in the table(s)?

```
dfs = pd.read_html('https://en.wikipedia.org/wiki/FIFA_World_Cup',
                        match='attendance')
print(len(dfs))
data = dfs[0]
print("Number of rows =",data.shape[0], "and number of columns=",data.shape[1])
```

```
1
Number of rows = 22 and number of columns= 9
```

3.2.5 Reading JSON data

JSON stands for JavaScript Object Notation, in which the data is stored and transmitted as plain text. A couple of benefits of the JSON format are:

1. Since the format is text only, JSON data can easily be exchanged between web applications, and used by any programming language.
2. Unlike the *csv* format, JSON supports a hierarchical data structure, and is easier to integrate with APIs.

The JSON format can support a hierachical data structure, as it is built on the following two data structures (*Source: technical documentation*):

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

The *Pandas* function read_json converts a JSON string to a Pandas DataFrame. The function *dumps()* of the *json* library converts a Python object to a JSON string.

Lets read the JSON data on Ted Talks.

```
tedtalks_data = pd.read_json('https://raw.githubusercontent.com/cwkenwaysun/TEDmap/master/
```

```
tedtalks_data.head()
```

	id	speaker	headline	URL	descripti
0	7	David Pogue	Simplicity sells	http://www.ted.com/talks/view/id/7	New Yor
1	6	Craig Venter	Sampling the ocean's DNA	http://www.ted.com/talks/view/id/6	Genomic
2	4	Burt Rutan	The real future of space exploration	http://www.ted.com/talks/view/id/4	In this p
3	3	Ashraf Ghani	How to rebuild a broken state	http://www.ted.com/talks/view/id/3	Ashraf G
4	5	Chris Bangle	Great cars are great art	http://www.ted.com/talks/view/id/5	America

3.2.6 Practice exercise 6

Read the movies dataset from here. How many rows and columns are there in the data?

```
movies_data = pd.read_json('https://raw.githubusercontent.com/vega/vega-datasets/master/da
print("Number of rows =",movies_data.shape[0], "and number of columns=",movies_data.shape[
```

```
Number of rows = 3201 and number of columns= 16
```

3.2.7 Reading data from web APIs

API, an acronym for Application programming interface, is a way of transferring information between systems. Many websites have public APIs that provide data via JSON or other formats. For example, the IMDb-API is a web service for receiving movies, serial, and cast information. API results are in the JSON format and include items such as movie specifications, ratings, Wikipedia page content, etc. One of these APIs contains ratings of the top 250 movies on IMDB. Let us read this data using the IMDB API.

We'll use the *get* function from the python library *requests* to request data from the API and obtain a response code. The response code will let us know if our request to pull data from this API was successful.

```
#Importing the requests library
import requests as rq
```

```
# Downloading imdb top 250 movie's data
url = 'https://imdb-api.com/en/API/Top250Movies/k_v6gf8ppf' #URL of the API containing top
```

```
response = rq.get(url) #Requesting data from the API
response
```

```
<Response [200]>
```

We have a response code of 200, which indicates that the request was successful.

The response object's JSON method will return a dictionary containing JSON parsed into native Python objects.

```
movie_data = response.json()
```

```
movie_data.keys()
```

```
dict_keys(['items', 'errorMessage'])
```

The *movie_data* contains only two keys. The *items* key seems likely to contain information about the top 250 movies. Let us convert the values of the *items* key (which is list of dictionaries) to a dataframe, so that we can view it in a tabular form.

```
#Converting a list of dictionaries to a dataframe
movie_data_df = pd.DataFrame(movie_data['items'])
```

```
#Checking the movie data pulled using the API
movie_data_df.head()
```

	id	rank	title	fullTitle	year	image
0	tt0111161	1	The Shawshank Redemption	The Shawshank Redemption (1994)	1994	https://m.r
1	tt0068646	2	The Godfather	The Godfather (1972)	1972	https://m.r
2	tt0468569	3	The Dark Knight	The Dark Knight (2008)	2008	https://m.r
3	tt0071562	4	The Godfather Part II	The Godfather Part II (1974)	1974	https://m.r
4	tt0050083	5	12 Angry Men	12 Angry Men (1957)	1957	https://m.r

```
#Rows and columns of the movie data
movie_data_df.shape
```

```
(250, 9)
```

This API provides the names of the top 250 movies along with the year of release, IMDB ratings, and cast information.

3.3 Writing data

The *Pandas* function *to_csv* can be used to write (or export) data to a *csv* or *txt* file. Below are some examples.

Example 1: Let us export the movies data of the top 250 movies to a *csv* file.

```
#Exporting the data of the top 250 movies to a csv file
movie_data_df.to_csv('movie_data_exported.csv')
```

The file *movie_data_exported.csv* will appear in the working directory.

Example 2: Let us export the movies data of the top 250 movies to a *txt* file with a semi-colon as the delimiter.

```
movie_data_df.to_csv('movie_data_exported.txt',sep=';')
```

Example 3: Let us export the movies data of the top 250 movies to a *JSON* file.

```
with open('movie_data.json', 'w') as f:
    json.dump(movie_data, f)
```

4 NumPy

```
&lt;IPython.core.display.Image object&gt;
```

NumPy, short for Numerical Python is used to analyze numeric data with Python. NumPy arrays are primarily used to create homogeneous n-dimensional arrays ($n = 1, ..., n$). Let us import the NumPy library to use its methods and functions, and the NumPy function *array()* to define a NumPy array.

```
import numpy as np
```

```
numpy_array = np.array([[1,2],[3,4]])
numpy_array
```

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

```
type(numpy_array)
```

```
numpy.ndarray
```

The NumPy function *array()* creates an object of type `numpy.ndarray`.

4.1 Why do we need NumPy arrays?

NumPy arrays can store data just like other data stuctures such as such as lists, tuples, and Pandas DataFrame. Computations performed using NumPy arrays can also be performed with data stored in the other data structures. However, NumPy is preferred for its efficiency, especially when working with large arrays of data.

4.1.1 Numpy arrays are memory efficient

A NumPy array is a collection of homogeneous data-types that are stored in contiguous memory locations. On the other hand, data structures such as lists are a collection of heterogeneous data types stored in non-contiguous memory locations. Homogenous data elements let the NumPy array be densely packed resulting in lesser memory consumption. The following example illustrates the smaller size of NumPy arrays as compared to other data structures.

```
#Example showing NumPy arrays take less storage space than lists, tuples and Pandas DataFr
tuple_ex = tuple(range(1000))
list_ex = list(range(1000))
numpy_ex = np.array([range(1000)])
pandas_df = pd.DataFrame(range(1000))
print("Space taken by tuple =",tuple_ex.__sizeof__()," bytes")
print("Space taken by list =",list_ex.__sizeof__()," bytes")
print("Space taken by Pandas DataFrame =",pandas_df.__sizeof__()," bytes")
print("Space taken by NumPy array =",numpy_ex.__sizeof__()," bytes")
```

```
Space taken by tuple = 8024  bytes
Space taken by list = 8040  bytes
Space taken by Pandas DataFrame = 8128  bytes
Space taken by NumPy array = 4120  bytes
```

Note that NumPy arrays are memory efficient as long as they are homogenous. They will lose the memory efficiency if they are used to store elements of multiple data types.

The example below compares the size of a homogenous NumPy array to that of a similar heterogenous NumPy array to illustrate the point.

```
numpy_homogenous = np.array([[1,2],[3,3]])
print("Size of a homogenous numpy array = ",numpy_homogenous.__sizeof__(), "bytes")
```

```
Size of homogenous numpy array =  136 bytes
```

Now let us convert an element of the above array to a string, and check the size of the array.

```
numpy_homogenous = np.array([[1,'2'],[3,3]])
print("Size of a heterogenous numpy array = ",numpy_homogenous.__sizeof__(), "bytes")
```

```
Size of a heterogenous numpy array =  296 bytes
```

The size of the homogenous NumPy array is much lesser than that of the one with heterogenous data. Thus, NumPy arrays are primarily used for storing homogenous data.

On the other hand, the size of other data structures, such as a list, does not depend on whether the elements in them are homogenous or heterogenous, as shown by the example below.

```
list_homogenous = list([1,2,3,4])
print("Size of a homogenous list = ",list_homogenous.__sizeof__(), "bytes")
list_heterogenous = list([1,'2',3,4])
print("Size of a heterogenous list = ",list_heterogenous.__sizeof__(), "bytes")
```

```
Size of a homogenous list =  72 bytes
Size of a heterogenous list =  72 bytes
```

Note that the memory efficiency of NumPy arrays does not come into play with a very small amount of data. Thus, a list with four elements - 1,2,3 and 4, has a lesser size than a NumPy array with the same elements. However, with larger datasets, such as the one shown earlier (sequence of integers from 0 to 999), the memory efficiency of NumPy arrays can be seen.

Unlike data structures such as lists, tuples, and dictionary, all elements of a NumPy array should be of same type to leverage the memory efficiency of NumPy arrays.

4.1.2 NumPy arrays are fast

With NumPy arrays, mathematical computations can be performed faster, as compared to other data structures, due to the following reasons:

1. As the NumPy array is **densely packed** with homogenous data, it helps retrieve the data faster as well, thereby making computations faster.
2. With NumPy, **vectorized computations** can replace the relatively more expensive python `for` loops. The NumPy package breaks down the vectorized computations into multiple fragments and then processes all the fragments parallelly. However, with a `for` loop, computations will be one at a time.
3. The NumPy package **integrates C**, and **C++** codes in Python. These programming languages have very little execution time as compared to Python.

We'll see the faster speed on NumPy computations in the example below.

Example: This example shows that computations using NumPy arrays are typically much faster than computations with other data structures.

Q: Multiply whole numbers upto 1 million by an integer, say 2. Compare the time taken for the computation if the numbers are stored in a NumPy array vs a list.

Use the numpy function arange() to define a one-dimensional NumPy array.

```
#Examples showing NumPy arrays are more efficient for numerical computation
import time as tm
start_time = tm.time()
list_ex = list(range(1000000)) #List containinig whole numbers upto 1 million
a=(list_ex*2)
print("Time take to multiply numbers in a list = ", tm.time()-start_time)

start_time = tm.time()
tuple_ex = tuple(range(1000000)) #Tuple containinig whole numbers upto 1 million
a=(tuple_ex*2)
print("Time take to multiply numbers in a tuple = ", tm.time()-start_time)

start_time = tm.time()
df_ex = pd.DataFrame(range(1000000)) #Pandas DataFrame containinig whole numbers upto 1 mi
a=(df_ex*2)
print("Time take to multiply numbers in a Pandas DataFrame = ", tm.time()-start_time)

start_time = tm.time()
numpy_ex = np.arange(1000000) #NumPy array containinig whole numbers upto 1 million
a=(numpy_ex*2)
print("Time take to multiply numbers in a NumPy array = ", tm.time()-start_time)
```

```
Time take to multiply numbers in a list =  0.023949384689331055
Time take to multiply numbers in a tuple =  0.03192734718322754
Time take to multiply numbers in a Pandas DataFrame =  0.047330617904663086
Time take to multiply numbers in a NumPy array =  0.0
```

4.2 NumPy array: Basic attributes

Let us define a NumPy array:

```
numpy_ex = np.array([[1,2,3],[4,5,6]])
numpy_ex
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

The attributes of `numpy_ex` can be seen by typing `numpy_ex` followed by a `.`, and then pressing the *tab* key.

Some of the basic attributes of a NumPy array are the following:

1. `ndim`: Shows the number of dimensions (or axes) of the array.

```
numpy_ex.ndim
```

```
2
```

2. `shape`: This is a tuple of integers indicating the size of the array in each dimension. For a matrix with *n* rows and *m* columns, the shape will be *(n,m)*. The length of the shape tuple is therefore the rank, or the number of dimensions, `ndim`.

```
numpy_ex.shape
```

```
(2, 3)
```

3. `size`: This is the total number of elements of the array, which is the product of the elements of shape.

```
numpy_ex.size
```

```
6
```

4. `dtype`: This is an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. NumPy provides many, for example bool_, character, int_, int8, int16, int32, int64, float_, float8, float16, float32, float64, complex_, complex64, object_.

```
numpy_ex.dtype
```

```
dtype('int32')
```

4.3 Arithematic operations

Numpy arrays support arithmetic operators like +, -, *, etc. We can perform an arithmetic operation on an array either with a single number (also called scalar) or with another array of the same shape. However, we cannot perform an arithematic operation on an array with an array of a different shape.

Below are some examples of arithematic operations on arrays.

```
#Defining two arrays of the same shape
arr1 = np.array([[1, 2, 3, 4],
                 [5, 6, 7, 8],
                 [9, 1, 2, 3]])
arr2 = np.array([[11, 12, 13, 14],
                 [15, 16, 17, 18],
                 [19, 11, 12, 13]])
```

```
#Element-wise summation of arrays
arr1 + arr2
```

```
array([[12, 14, 16, 18],
       [20, 22, 24, 26],
       [28, 12, 14, 16]])
```

```
# Element-wise subtraction
arr2 - arr1
```

```
array([[10, 10, 10, 10],
       [10, 10, 10, 10],
       [10, 10, 10, 10]])
```

```
# Adding a scalar to an array adds the scalar to each element of the array
arr1 + 3
```

```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12,  4,  5,  6]])
```

```python
# Dividing an array by a scalar divides all elements of the array by the scalar
arr1 / 2
```

```
array([[0.5, 1. , 1.5, 2. ],
       [2.5, 3. , 3.5, 4. ],
       [4.5, 0.5, 1. , 1.5]])
```

```python
# Element-wise multiplication
arr1 * arr2
```

```
array([[ 11,  24,  39,  56],
       [ 75,  96, 119, 144],
       [171,  11,  24,  39]])
```

```python
# Modulus operator with scalar
arr1 % 4
```

```
array([[1, 2, 3, 0],
       [1, 2, 3, 0],
       [1, 1, 2, 3]], dtype=int32)
```

4.4 Broadcasting

Broadcasting allows arithmetic operations between two arrays with different numbers of dimensions but compatible shapes.

The Broadcasting documentation succinctly explains it as the following:

"The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is *broadcast* across the larger array so that they have compatible shapes. Broadcasting provides a means of vectorizing array operations so that looping occurs in C instead of Python. It does this without making needless copies of data and usually leads to efficient algorithm implementations."

The example below shows the broadcasting of two arrays.

```python
arr1 = np.array([[1, 2, 3, 4],
                 [5, 6, 7, 8],
                 [9, 1, 2, 3]])
```

```python
arr2 = np.array([4, 5, 6, 7])
```

```python
arr1 + arr2
```

```
array([[ 5,  7,  9, 11],
       [ 9, 11, 13, 15],
       [13,  6,  8, 10]])
```

When the expression `arr1 + arr2` is evaluated, `arr2` (which has the shape `(4,)`) is replicated three times to match the shape `(3, 4)` of `arr1`. Numpy performs the replication without actually creating three copies of the smaller dimension array, thus improving performance and using lower memory.

In the above addition of arrays, `arr2` was *stretched* or *broadcast* to the shape of `arr1`. However, this broadcasting was possible only because the right dimension of both the arrays is 4, and the left dimension of one of the arrays is 1.

See the broadcasting documentation to understand the rules for broadcasting:

"When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimensions and works its way left. Two dimensions are compatible when:

- they are equal, or
- one of them is 1"

If the rightmost dimension of `arr2` is 3, broadcasting will not occur, as it is not equal to the rightmost dimension of `arr1`:

```python
#Defining arr2 as an array of shape (3,)
arr2 = np.array([4, 5, 6])
```

```python
#Broadcasting will not happen when the broadcasting rules are violated
arr1 + arr2
```

```
ValueError: operands could not be broadcast together with shapes (3,4) (3,)
```

4.5 Comparison

Numpy arrays support comparison operations like `==`, `!=`, `>` etc. The result is an array of booleans.

```
arr1 = np.array([[1, 2, 3], [3, 4, 5]])
arr2 = np.array([[2, 2, 3], [1, 2, 5]])
```

```
arr1 == arr2
```

```
array([[False,  True,  True],
       [False, False,  True]])
```

```
arr1 != arr2
```

```
array([[ True, False, False],
       [ True,  True, False]])
```

```
arr1 >= arr2
```

```
array([[False,  True,  True],
       [ True,  True,  True]])
```

```
arr1 < arr2
```

```
array([[ True, False, False],
       [False, False, False]])
```

Array comparison is frequently used to count the number of equal elements in two arrays using the `sum` method. Remember that `True` evaluates to `1` and `False` evaluates to `0` when booleans are used in arithmetic operations.

```
(arr1 == arr2).sum()
```

```
3
```

4.6 Concatenating arrays

Arrays can be concatenated along an axis with NumPy's concatenate function. The `axis` argument specifies the dimension for concatenation. The arrays should have the same number of dimensions, and the same length along each axis except the axis used for concatenation.

The examples below show concatenation of arrays.

```
arr1 = np.array([[1, 2, 3], [3, 4, 5]])
arr2 = np.array([[2, 2, 3], [1, 2, 5]])
print("Array 1:\n",arr1)
print("Array 2:\n",arr2)
```

```
Array 1:
 [[1 2 3]
 [3 4 5]]
Array 2:
 [[2 2 3]
 [1 2 5]]
```

```
#Concatenating the arrays along the default axis: axis=0
np.concatenate((arr1,arr2))
```

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

```
#Concatenating the arrays along axis = 1
np.concatenate((arr1,arr2),axis=1)
```

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

Since the arrays need to have the same dimension only along the axis of concatenation, let us try concatenate the array below (`arr3`) with `arr1`, along axis = 0.

```
arr3 = np.array([2, 2, 3])
```

```
np.concatenate((arr1,arr3),axis=0)
```

```
ValueError: all the input arrays must have same number of dimensions, but the array at index
```

Note the above error, which indicates that `arr3` has only one dimension. Let us check the shape of `arr3`.

```
arr3.shape
```

```
(3,)
```

We can reshape `arr3` to a shape of (1,3) to make it compatible for concatenation with `arr1` along axis = 0.

```
arr3_reshaped = arr3.reshape(1,3)
arr3_reshaped
```

```
array([[2, 2, 3]])
```

Now we can concatenate the reshaped `arr3` with `arr1` along axis = 0.

```
np.concatenate((arr1,arr3_reshaped),axis=0)
```

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

4.7 Practice exercise 1

4.7.0.1

Read the coordinates of the capital cities of the world from http://techslides.com/list-of-countries-and-capitals . Use NumPy to print the name and coordinates of the capital city closest to the US capital - Washington DC.

Note that:

1. The *Country Name* for US is given as *United States* in the data.
2. The 'closeness' of capital cities from the US capital is based on the Euclidean distance of their coordinates to those of the US capital.

Hints:

1. Use *read_html()* from the *Pandas* library to read the table.
2. Use the *to_numpy()* function of the *Pandas DataFrame* class to convert a DataFrame to a Numpy array

3. Use *broadcasting* to compute the euclidean distance of capital cities from Washington DC.

Solution:

```
import pandas as pd
capital_cities = pd.read_html('http://techslides.com/list-of-countries-and-capitals',heade
coordinates_capital_cities = capital_cities.iloc[:,2:4].to_numpy()
us_coordinates = capital_cities.loc[capital_cities['Country Name']=='United States',['Capi

#Broadcasting
distance_from_DC = np.sqrt(np.sum((us_coordinates-coordinates_capital_cities)**2,axis=1))

#Assigning a high value of distance to DC, otherwise it will itself be selected as being c
distance_from_DC[distance_from_DC==0]=9999
closest_capital_index = np.argmin(distance_from_DC)
print("Closest capital city is:" ,capital_cities.loc[closest_capital_index,'Capital Name']
print("Coordinates of the closest capital city are:",coordinates_capital_cities[closest_ca
```

```
Closest capital city is: Ottawa
Coordinates of the closest capital city are: [ 45.41666667 -75.7        ]
```

4.7.0.2

Use NumPy to:

1. Print the names of the countries of the top 10 capital cities closest to the US capital - Washington DC.
2. Create and print a NumPy array containing the coordinates of the top 10 cities.

Hint: Use the *concatenate()* function from the *NumPy* library to stack the coordinates of the top 10 cities.

```
top10_cities_coordinates = coordinates_capital_cities[closest_capital_index,:].reshape(1,2
print("Top 10 countries closest to Washington DC are:\n Canada")
for i in range(9):
    distance_from_DC[closest_capital_index]=9999
    closest_capital_index = np.argmin(distance_from_DC)
    print(capital_cities.loc[closest_capital_index,'Country Name'])
    top10_cities_coordinates=np.concatenate((top10_cities_coordinates,coordinates_capital_
print("Coordinates of the top 10 cities closest to US are: \n",top10_cities_coordinates)
```

```
Top 10 countries closest to Washington DC are:
 Canada
Bahamas
Bermuda
Cuba
Turks and Caicos Islands
Cayman Islands
Haiti
Jamaica
Dominican Republic
Saint Pierre and Miquelon
Coordinates of the top 10 cities closest to US are:
 [[ 45.41666667 -75.7       ]
 [ 25.08333333 -77.35      ]
 [ 32.28333333 -64.783333  ]
 [ 23.11666667 -82.35      ]
 [ 21.46666667 -71.133333  ]
 [ 19.3        -81.383333  ]
 [ 18.53333333 -72.333333  ]
 [ 18.         -76.8       ]
 [ 18.46666667 -69.9       ]
 [ 46.76666667 -56.183333  ]]
```

4.8 Vectorized computation with NumPy

Several matrix algebra operations such as multiplications, decompositions, determinants, etc. can be performed conveniently with NumPy. However, we'll focus on matrix multiplication as it is very commonly used to avoid python `for` loops and make computations faster. The dot function is used to multiply matrices:

```
#Defining a 2x2 matrix
a = np.array([[0,1],[3,4]])
a
```

```
array([[0, 1],
       [3, 4]])
```

```
#Defining a 2x2 matrix
b = np.array([[6,-1],[2,1]])
b
```

```
array([[ 6, -1],
       [ 2,  1]])
```

```
#Multiplying matrices 'a' and 'b' using the dot function
a.dot(b)
```

```
array([[ 2,  1],
       [26,  1]])
```

```
#Note that * results in element-wise multiplication
a*b
```

```
array([[ 0, -1],
       [ 6,  4]])
```

Example 2: This example will show vectorized computations with NumPy. Vectorized computations help perform computations more efficiently, and also make the code concise.

Q: Read the (1) quantities of roll, bun, cake and bread required by 3 people - Ben, Barbara & Beth, from *food_quantity.csv*, (2) price of these food items in two shops - Target and Kroger, from *price.csv*. Find out which shop should each person go to minimize their expenses.

```
#Reading the datasets on food quantity and price
import pandas as pd
food_qty = pd.read_csv('./Datasets/food_quantity.csv')
price = pd.read_csv('./Datasets/price.csv')
```

```
food_qty
```

	Person	roll	bun	cake	bread
0	Ben	6	5	3	1
1	Barbara	3	6	2	2
2	Beth	3	4	3	1

```
price
```

	Item	Target	Kroger
0	roll	1.5	1.0
1	bun	2.0	2.5
2	cake	5.0	4.5
3	bread	16.0	17.0

First, let's start from a simple problem. We'll compute the expenses of Ben if he prefers to buy all food items from Target

```
#Method 1: Using loop
bens_target_expense = 0 #Initializing Ben's expenses to 0
for k in range(4):   #Iterating over all the four desired food items
    bens_target_expense += food_qty.iloc[0,k+1]*price.iloc[k,1] #Total expenses on the kth
bens_target_expense    #Total expenses for Ben if he goes to Target
```

```
50.0
```

```
#Method 2: Using NumPy array
food_num = food_qty.iloc[0,1:].to_numpy()  #Converting food quantity (for Ben) dataframe t
price_num = price.iloc[:,1].to_numpy()     #Converting price (for Target) dataframe to Num
food_num.dot(price_num)   #Matrix multiplication of the quantity vector with the price vec
```

```
50.0
```

Ben will spend $50 if he goes to Target

Now, let's add another layer of complication. We'll compute Ben's expenses for both stores - Target and Kroger

```
#Method 1: Using loops

#Initializing a Series of length two to store the expenses in Target and Kroger for Ben
bens_store_expense = pd.Series(0.0,index=price.columns[1:3])
for j in range(2):      #Iterating over both the stores - Target and Kroger
    for k in range(4):        #Iterating over all the four desired food items
        bens_store_expense[j] += food_qty.iloc[0,k+1]*price.iloc[k,j+1]
bens_store_expense
```

```
Target    50.0
Kroger    49.0
dtype: float64
```

```
#Method 2: Using NumPy array
food_num = food_qty.iloc[0,1:].to_numpy()  #Converting food quantity (for Ben) dataframe t
price_num = price.iloc[:,1:].to_numpy()    #Converting price dataframe to NumPy array
food_num.dot(price_num)      #Matrix multiplication of the quantity vector with the price
```

```
array([50.0, 49.0], dtype=object)
```

Ben will spend \$50 if he goes to Target, and $49 if he goes to Kroger. Thus, he should choose Kroger.

Now, let's add the final layer of complication, and solve the problem. We'll compute everyone's expenses for both stores - Target and Kroger

```
#Method 1: Using loops
store_expense = pd.DataFrame(0.0,index=price.columns[1:3],columns = food_qty['Person'])
for i in range(3):    #Iterating over all the three people - Ben, Barbara, and Beth
    for j in range(2):     #Iterating over both the stores - Target and Kroger
        for k in range(4):        #Iterating over all the four desired food items
            store_expense.iloc[j,i] += food_qty.iloc[i,k+1]*price.iloc[k,j+1]
store_expense
```

Person	Ben	Barbara	Beth
Target	50.0	58.5	43.5
Kroger	49.0	61.0	43.5

```
#Method 2: Using NumPy array
food_num = food_qty.iloc[:,1:].to_numpy() #Converting food quantity dataframe to NumPy arr
price_num = price.iloc[:,1:].to_numpy()  #Converting price dataframe to NumPy array
food_num.dot(price_num)  #Matrix multiplication of the quantity matrix with the price matr
```

```
array([[50. , 49. ],
       [58.5, 61. ],
       [43.5, 43.5]])
```

Based on the above table, Ben should go to Kroger, Barbara to Target and Beth can go to either store.
Note that, with each layer of complication, the number of for loops keep increasing, thereby increasing the complexity of Method 1, while the method with NumPy array does not change much. Vectorized computations with arrays are much more efficient.

4.8.1 Practice exercise 2

Use matrix multiplication to find the average IMDB rating and average Rotten tomatoes rating for each genre - comedy, action, drama and horror. Use the data: *movies_cleaned.csv*. Which is the most preferred genre for IMDB users, and which is the least preferred genre for Rotten Tomatoes users?

Hint: 1. Create two matrices - one containing the IMDB and Rotten Tomatoes ratings, and the other containing the genre flags (comedy/action/drama/horror).

2. Multiply the two matrices created in 1.
3. Divide each row/column of the resulting matrix by a vector having the number of ratings in each genre to get the average rating for the genre.

4.9 Pseudorandom number generation

Random numbers often need to be generated to analyze processes or systems, especially in cases when these processes or systems are governed by known probability distrbutions. For example, the number of personnel required to answer calls at a call center can be analyzed by simulating occurence and duration of calls.

NumPy's random module can be used to generate arrays of random numbers from several different probability distributions. For example, a `3x5` array of uniformly distributed random numbers can be generated using the `uniform` function of the `random` module.

```
np.random.uniform(size = (3,5))
```

```
array([[0.69256322, 0.69259973, 0.03515058, 0.45186048, 0.43513769],
       [0.07373366, 0.07465425, 0.92195975, 0.72915895, 0.8906299 ],
       [0.15816734, 0.88144978, 0.05954028, 0.81403832, 0.97725557]])
```

Random numbers can also be generated by Python's built-in random module. However, it generates one random number at a time, which makes it much slower than NumPy's random module.

Example: Suppose 500 people eat at Food cart 1, and another 500 eat at Food cart 2, everyday.

The waiting time at Food cart 2 has a normal distribution with mean 8 minutes and standard deviation 3 minutes, while the waiting time at Food cart 1 has a uniform distribution with minimum 5 minutes and maximum 25 minutes.

Simulate a dataset containing waiting times for 500 ppl for 30 days in each of the food joints. Assume that the waiting times are measured simultaneously at a certain time in both places, i.e., the observations are paired.

On how many days is the average waiting time at Food cart 2 higher than that at Food cart 1?

What percentage of times the waiting time at Food cart 2 was higher than the waiting time at Food cart 1?

Try both approaches: (1) Using loops to generate data, (2) numpy array to generate data. Compare the time taken in both approaches.

```
import time as tm
```

```
#Method 1: Using loops
start_time = tm.time() #Current system time

#Initializing waiting times for 500 ppl over 30 days
waiting_times_FoodCart1 = pd.DataFrame(0,index=range(500),columns=range(30)) #FoodCart1
waiting_times_FoodCart2 = pd.DataFrame(0,index=range(500),columns=range(30)) #FoodCart2
import random as rm
for i in range(500):  #Iterating over 500 ppl
    for j in range(30): #Iterating over 30 days
        waiting_times_FoodCart2.iloc[i,j] = rm.gauss(8,3) #Simulating waiting time in Food
        waiting_times_FoodCart1.iloc[i,j] = rm.uniform(5,25) #Simulating waiting time in F
time_diff = waiting_times_FoodCart2-waiting_times_FoodCart1

print("On ",sum(time_diff.mean()>0)," days, the average waiting time at FoodCart2 higher t
print("Percentage of times waiting time at FoodCart2 was greater than that at FoodCart1 =
end_time = tm.time() #Current system time
print("Time taken = ", end_time-start_time)
```

```
On  0  days, the average waiting time at FoodCart2 higher than that at FoodCart1
Percentage of times waiting time at FoodCart2 was greater than that at FoodCart1 =  16.226666
Time taken =  4.521248817443848
```

```
#Method 2: Using NumPy arrays
start_time = tm.time()
waiting_time_FoodCart2 = np.random.normal(8,3,size = (500,30)) #Simultaneously generating
waiting_time_FoodCart1 = np.random.uniform(5,25,size = (500,30)) #Simultaneously generatin
time_diff = waiting_time_FoodCart2-waiting_time_FoodCart1
print("On ",(time_diff.mean()>0).sum()," days, the average waiting time at FoodCart2 highe
print("Percentage of times waiting time at FoodCart2 was greater than that at FoodCart1 =
end_time = tm.time()
print("Time taken = ", end_time-start_time)
```

```
On  0  days, the average waiting time at FoodCart2 higher than that at FoodCart1
Percentage of times waiting time at FoodCart2 was greater than that at FoodCart1 =  16.52 %
Time taken =  0.008000850677490234
```

The approach with NumPy is much faster than the one with loops.

4.9.1 Practice exercise 3

Bootstrapping: Find the 95% confidence interval of mean profit for 'Action' movies, using Bootstrapping.

Bootstrapping is a non-parametric method for obtaining confidence interval. Use the algorithm below to find the confidence interval:

1. Find the profit for each of the 'Action' movies. Suppose there are N such movies. We will have a *Profit* column with N values.

2. Randomly sample N values with replacement from the *Profit* column

3. Find the mean of the N values obtained in (b)

4. Repeat steps (b) and (c) *M=1000* times

5. The 95% Confidence interval is the range between the 2.5% and 97.5% percentile values of the 1000 means obtained in (c)
   Use the *movies_cleaned.csv* dataset.

5 Pandas

The Pandas library contains several methods and functions for cleaning, manipulating and analyzing data. While NumPy is suited for working with homogenous numerical array data, Pandas is designed for working with tabular or heterogenous data.

Pandas is built on top of the NumPy package. Thus, there are some similarities between the two libraries. Like NumPy, Pandas provides the basic mathematical functionalities like addition, subtraction, conditional operations and broadcasting. However, unlike NumPy library which provides objects for multi-dimensional arrays, Pandas provides the 2D table object called Dataframe.

Data in pandas is often used to feed statistical analysis in SciPy, plotting functions from Matplotlib, and machine learning algorithms in Scikit-learn.

Typically, the Pandas library is used for:

- Cleaning the data by tasks such as removing missing values, filtering rows / columns, aggregating data, mutating data, etc.
- Computing summary statistics such as the mean, median, max, min, standard deviation, etc.
- Computing correlation among columns in the data
- Computing the data distribtion
- Visualizing the data with help from the Matplotlib library
- Storing the cleaned, transformed data back into a CSV, other file or database

Let's import the Pandas library to use its methods and functions.

```
import pandas as pd
```

5.1 Pandas data stuctures - Series and DataFrame

There are two core components of the Pandas library - Series and DataFrame.

A DataFrame is a two-dimensional object - conmprising of tabular data organized in rows and columns, where individual columns can be of different value types (numeric / string / boolean etc.). A DataFrame has row lables (also called row indices) which refer to individual rows, and column lables (also called column names) that refer to indvidual columns. By default, the row indices are integers starting from zero. However, both the row indices and column names can be customized by the user.

Let us read the spotify data - *spotify_data.csv*, using the Pandas function `read_csv()`.

```
spotify_data = pd.read_csv('./Datasets/spotify_data.csv')
spotify_data.head()
```

	artist_followers	genres	artist_name	artist_popularity	track_name	track_popularity
0	16996777	rap	Juice WRLD	96	All Girls Are The Same	0
1	16996777	rap	Juice WRLD	96	Lucid Dreams	0
2	16996777	rap	Juice WRLD	96	Hear Me Calling	0
3	16996777	rap	Juice WRLD	96	Robbery	0
4	5988689	rap	Roddy Ricch	88	Big Stepper	0

The object `spotify_data` is a pandas DataFrame:

```
type(spotify_data)
```

```
pandas.core.frame.DataFrame
```

A Series is a one-dimensional object, containing a sequence of values, where each value has an index. Each column of a DataFrame is Series as shown in the example below.

```
#Extracting movie titles from the movie_ratings DataFrame
spotify_songs = movie_ratings['track_name']
spotify_songs
```

```
0                       Eno Ide
1             Ee Tanuvu Ninnade
2             Munjaane Manjalli
3          Gudugudiya Sedi Nodo
4                         Ambar
                  ...
245618          Coming Up Roses
245619                Young Kid
245620                 Apricots
245621       Time I Love to Waste
245622                  Call me
Name: track_name, Length: 245623, dtype: object
```

```
#The object movie_titles is a Series
type(spotify_songs)
```

```
pandas.core.series.Series
```

A Series is essentially a column, and a DataFrame is a two-dimensional table made up of a collection of Series

```
<IPython.core.display.Image object>
```

5.2 Creating a Pandas Series / DataFrame

5.2.1 Specifying data within the Series() / DataFrame() functions

A Pandas Series and DataFrame can be created by specifying the data within the Series() / DataFrame() function. Below are examples of defining a Pandas Series / DataFrame.

```
#Defining a Pandas Series
series_example = pd.Series(['these','are','english','words'])
series_example
```

```
0      these
1        are
2    english
3      words
dtype: object
```

Note that the default row indices are integers starting from 0. However, the index can be specified with the `index` argument if desired by the user:

```
#Defining a Pandas Series with custom row labels
series_example = pd.Series(['these','are','english','words'], index = range(101,105))
series_example
```

```
101      these
102        are
103    english
104      words
dtype: object
```

5.2.2 Transforming in-built data structures

A Pandas DataFrame can be created by converting the in-built python data structures such as lists, dictionaries, and list of dictionaries to DataFrame. See the examples below.

```
#List consisting of expected age to marry of students of the STAT303-1 Fall 2022 class
exp_marriage_age_list=['24','30','28','29','30','27','26','28','30+','26','28','30','30','
```

```
#Example 1: Creating a Pandas Series from a list
exp_marriage_age_series=pd.Series(exp_marriage_age_list,name = 'expected_marriage_age')
exp_marriage_age_series.head()
```

```
0    24
1    30
2    28
3    29
4    30
Name: expected_marriage_age, dtype: object
```

```
#Dictionary consisting of the GDP per capita of the US from 1960 to 2021 with some missing
GDP_per_capita_dict = {'1960':3007,'1961':3067,'1962':3244,'1963':3375,'1964':3574,'1965':
```

```
#Example 2: Creating a Pandas Series from a Dictionary
GDP_per_capita_series = pd.Series(GDP_per_capita_dict)
GDP_per_capita_series.head()
```

```
1960    3007
1961    3067
1962    3244
1963    3375
1964    3574
dtype: int64
```

```
#List of dictionary consisting of 52 playing cards of the deck
deck_list_of_dictionaries = [{'value':i, 'suit':c}
for c in ['spades', 'clubs', 'hearts', 'diamonds']
for i in range(2,15)]
```

```
#Example 3: Creating a Pandas DataFrame from a List of dictionaries
deck_df = pd.DataFrame(deck_list_of_dictionaries)
deck_df.head()
```

	value	suit
0	2	spades
1	3	spades
2	4	spades
3	5	spades
4	6	spades

5.2.3 Importing data from files

In the real world, a Pandas DataFrame will typically be created by loading the datasets from existing storage such as SQL Database, CSV file, Excel file, text files, HTML files, etc., as we learned in the third chapter of the book on Reading data.

5.3 Attributes of a Pandas DataFrame

The Pandas DataFrame class has the following commonly used attributes.

1. dtypes: It lists the type of columns.

```
spotify_data.dtypes
```

```
artist_followers       int64
genres                object
artist_name           object
artist_popularity      int64
track_name            object
track_popularity       int64
duration_ms            int64
explicit               int64
release_year           int64
danceability         float64
energy               float64
key                    int64
loudness             float64
```

```
mode                    int64
speechiness           float64
acousticness          float64
instrumentalness      float64
liveness              float64
valence               float64
tempo                 float64
time_signature          int64
dtype: object
```

Table **?@tbl-dtype** describes the datatypes of columns in a Pandas DataFrame.

Pandas Type	Native Python Type	Description
object	string	The most general dtype. This datatype is assigned to a column if the column has mixed types (numbers and strings)
int64	int	This datatype is for integers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 or for integers havind a maximum sizeo of 64 bits
float64	float	This datatype is for real numbers. If a column contains integers and NaNs, Pandas will default to float64. This is because the missing values may be a real number
datetime64, timedelta[ns]	N/A (but see the datetime module in Python's standard library)	Values meant to hold time data. This datatype is useful for time series analysis

5.4 Data manipulations with Pandas

5.4.1 Sub-setting data

In the chapter on reading data, we learned about operators `loc` and `iloc` that can be used to subset data based on axis labels and position of rows/columns respectively. However, usually we are not aware of the relevant row indices, and we may want to subset data based on some condition(s). For example, suppose we wish to analyze only those songs whose track popularity is higher than 50.

Q: Do we need to subset rows or columns in this case?
A: Rows, as songs correspond to rows, while features of songs correspond to columns.

As we need to subset rows, the filter must be applied at the starting index. As we don't need to subset any specific features of the songs, there is no subsetting to be done on the columns. A : at the ending index means that all columns need to selected.

```
#Subsetting spotify songs that have track popularity score of more than 50
popular_songs = spotify_data.loc[spotify_data.track_popularity>=50,:]
popular_songs.head()
```

	artist_followers	genres	artist_name	artist_popularity	track_name	track
181	1277325	hip hop	Dave	77	Titanium	69
191	1123869	rap	Jay Wheeler	85	Viendo el Techo	64
208	3657199	rap	Polo G	91	RAPSTAR	89
263	1461700	pop & rock	Teoman	67	Gecenin Sonuna Yolculuk	52
293	299746	pop & rock	Lars Winnerbäck	62	Själ och hjärta	55

Suppose we wish to analyze only *track_name, release year* and *track_popularity* of songs. Then, we can subset the revelant columns:

```
relevant_columns = spotify_data.loc[:,['track_name','release_year','track_popularity']]
relevant_columns.head()
```

	track_name	release_year	track_popularity
0	All Girls Are The Same	2021	0
1	Lucid Dreams	2021	0
2	Hear Me Calling	2021	0
3	Robbery	2021	0
4	Big Stepper	2021	0

5.4.2 Sorting data

Sorting dataset is a very common operation. The sort_values() function of Pandas can be used to sort a Pandas DataFrame or Series. Let us sort the spotify data in decreasing order of *track_popularity*:

```
spotify_sorted = spotify_data.sort_values(by = 'track_popularity', ascending = False)
spotify_sorted.head()
```

	artist_followers	genres	artist_name	artist_popularity	track_name	track_popu
2398	1444702	pop	Olivia Rodrigo	88	drivers license	99
2442	177401	hip hop	Masked Wolf	85	Astronaut In The Ocean	98
3133	1698014	pop	Kali Uchis	88	telepatía	97
6702	31308207	pop	The Weeknd	96	Save Your Tears	97
6703	31308207	pop	The Weeknd	96	Blinding Lights	96

Drivers license is the most popular song!

```
<IPython.core.display.HTML object>
```

5.4.3 Unique values, value counts and membership

The Pandas function unique provides the unique values of a Series. For example, let us find the number of unique genres of songs in the spotify dataset:

```
spotify_data.genres.unique()
```

```
array(['rap', 'pop', 'miscellaneous', 'metal', 'hip hop', 'rock',
       'pop & rock', 'hoerspiel', 'folk', 'electronic', 'jazz', 'country',
       'latin'], dtype=object)
```

The Pandas function value_counts() provides the number of observations of each value of a Series. For example, let us find the number of songs of each genre in the spotify dataset:

```
spotify_data.genres.value_counts()
```

```
pop              70441
rock             49785
pop & rock       43437
miscellaneous    35848
jazz             13363
```

```
hoerspiel        12514
hip hop           7373
folk              2821
latin             2125
rap               1798
metal             1659
country           1236
electronic         790
Name: genres, dtype: int64
```

More than half the songs in the dataset are *pop, rock* or *pop & rock*.

The Pandas function isin() provides a boolean Series indicating the position of certain values in a Series. The function is helpful in sub-setting data. For example, let us subset the songs that are either *latin, rap,* or *metal*:

```
latin_rap_metal_songs = spotify_data.loc[spotify_data.genres.isin(['latin','rap','metal'])
latin_rap_metal_songs.head()
```

	artist_followers	genres	artist_name	artist_popularity	track_name	track_popularity
0	16996777	rap	Juice WRLD	96	All Girls Are The Same	0
1	16996777	rap	Juice WRLD	96	Lucid Dreams	0
2	16996777	rap	Juice WRLD	96	Hear Me Calling	0
3	16996777	rap	Juice WRLD	96	Robbery	0
4	5988689	rap	Roddy Ricch	88	Big Stepper	0

5.5 Operations between DataFrame and Series

Let us learn arithematic operations between DataFrame and Series with the help of an example.

Example: Spotify recommends songs based on songs listened by the user. Suppose you have listned to the song *drivers license*. Spotify intends to recommend you 5 songs that are *similar* to *drivers license*. Which songs should it recommend?

Let us see what information do we have about songs that can help us identify songs similar to drivers license. The columns attribute of DataFrame will display all the columns names. The description of some of the column names relating to audio features is here.

```
spotify_data.columns
```

```
Index(['artist_followers', 'genres', 'artist_name', 'artist_popularity',
       'track_name', 'track_popularity', 'duration_ms', 'explicit',
       'release_year', 'danceability', 'energy', 'key', 'loudness', 'mode',
       'speechiness', 'acousticness', 'instrumentalness', 'liveness',
       'valence', 'tempo', 'time_signature'],
      dtype='object')
```

Solution approach: We have serveral features of a song. Let us find songs similar to *drivers license* in terms of *danceability, energy, key, loudness, mode, speechiness, acousticness, instrumentalness, liveness, valence, time_signature* and *tempo*. Note that we are considering only audio features for simplicity.

To find the songs most similar to *drivers license*, we need to define a measure that quantifies the similarity. Let us define simililarity of a song with *drivers license* as the Euclidean distance of the song from *drivers license*, where the coordinates of a song are: (danceability, energy, key, loudness, mode, speechiness, acousticness, instrumentalness, liveness, valence, time_signature, tempo). Thus, similarity can be formulated as:

$$Similarity_{DL-S} = \sqrt{(danceability_{DL} - danceability_S)^2 + (energy_{DL} - energy_S)^2 + ... + (tempo_{DL} - tempo_S)$$

where the subscript DL stands for *drivers license* and S stands for any song. The top 5 songs with the least value of $Similarity_{DL-S}$ will be the most similar to *drivers lincense* and should be recommended.

Let us subset the columns that we need to use to compute the Euclidean distance.

```
audio_features = spotify_data[['danceability', 'energy', 'key', 'loudness','mode','speechi
                                'acousticness', 'instrumentalness', 'liveness','valence', '
```

```
audio_features.head()
```

	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	liveness	v
0	0.673	0.529	0	-7.226	1	0.3060	0.0769	0.000338	0.0856	0
1	0.511	0.566	6	-7.230	0	0.2000	0.3490	0.000000	0.3400	0
2	0.699	0.687	7	-3.997	0	0.1060	0.3080	0.000036	0.1210	0
3	0.708	0.690	2	-5.181	1	0.0442	0.3480	0.000000	0.2220	0
4	0.753	0.597	8	-8.469	1	0.2920	0.0477	0.000000	0.1970	0

```
#Distribution of values of audio_features
audio_features.describe()
```

	danceability	energy	key	loudness	mode	speechiness	ac
count	243190.000000	243190.000000	243190.000000	243190.000000	243190.000000	243190.000000	24
mean	0.568357	0.580633	5.240326	-9.432548	0.670928	0.111984	0.
std	0.159444	0.236631	3.532546	4.449731	0.469877	0.198068	0.
min	0.000000	0.000000	0.000000	-60.000000	0.000000	0.000000	0.
25%	0.462000	0.405000	2.000000	-11.990000	0.000000	0.033200	0.
50%	0.579000	0.591000	5.000000	-8.645000	1.000000	0.043100	0.
75%	0.685000	0.776000	8.000000	-6.131000	1.000000	0.075300	0.
max	0.988000	1.000000	11.000000	3.744000	1.000000	0.969000	0.

Note that the audio features differ in terms of scale. Some features like *key* have a wide range of [0,11], while others like *danceability* have a very narrow range of [0,0.988]. If we use them directly, features like *danceability* will have a much higher influence on $Similarity_{DL-S}$ as compared to features like *key*. Assuming we wish all the features to have equal weight in quantifying a song's similarity to *drivers license*, we should scale the features, so that their values are comparable.

Let us scale the value of each column to a standard uniform distribtion: $U[0,1]$.

For scaling the values of a column to $U[0,1]$, we need to subtract the minimum value of the column from each value, and divide by the range of values of the column. For example, *danceability* can be standardized as follows:

```
#Scaling danceability to U[0,1]
danceability_value_range = audio_features.danceability.max()-audio_features.danceability.m
danceability_std = (audio_features.danceability-audio_features.danceability.min())/danceab
danceability_std
```

```
0         0.681174
1         0.517206
2         0.707490
3         0.716599
4         0.762146
            ...
243185    0.621457
243186    0.797571
243187    0.533401
243188    0.565789
```

```
243189    0.750000
Name: danceability, Length: 243190, dtype: float64
```

However, it will be cumbersome to repeat the above code for each audio feature. We can instead write a function that scales values of a column to $U[0,1]$, and apply the function on all the audio features.

```
#Function to scale a column to U[0,1]
def scale_uniform(x):
    return (x-x.min())/(x.max()-x.min())
```

We will use the Pandas function apply() to apply the above function to the DataFrame `audio_features`.

```
#Scaling all audio features to U[0,1]
audio_features_scaled = audio_features.apply(scale_uniform)
```

lambda function: Note that one line functions can be conveniently written as lambda functions in Python. These functions do not require a name, and can be defined using the keyword `lambda`. The above two blocks of code can be concisely written as:

```
audio_features_scaled = audio_features.apply(lambda x: (x-x.min())/(x.max()-x.min()))
```

```
#All the audio features are scaled to U[0,1]
audio_features_scaled.describe()
```

	danceability	energy	key	loudness	mode	speechiness	ac
count	243190.000000	243190.000000	243190.000000	243190.000000	243190.000000	243190.000000	24
mean	0.575260	0.580633	0.476393	0.793290	0.670928	0.115566	0.
std	0.161380	0.236631	0.321141	0.069806	0.469877	0.204405	0.
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.
25%	0.467611	0.405000	0.181818	0.753169	0.000000	0.034262	0.
50%	0.586032	0.591000	0.454545	0.805644	1.000000	0.044479	0.
75%	0.693320	0.776000	0.727273	0.845083	1.000000	0.077709	0.
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.

Since we need to find the Euclidean distance from the song *drivers license*, let us find the index of the row containing features of *drivers license.

```
drivers_license_index = spotify_data[spotify_data.track_name=='drivers license'].index[0]
```

Now, we'll subtract the audio features of *drivers license* from all other songs:

```
songs_minus_DL = audio_features_scaled-audio_features_scaled.loc[drivers_license_index,:]
```

Now, let us square the difference computed above. We'll use the in-built python function `pow()` to square the difference:

```
songs_minus_DL_sq = songs_minus_DL.pow(2)
songs_minus_DL_sq.head()
```

	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	live
0	0.007933	0.008649	0.826446	0.000580	0.0	0.064398	0.418204	1.055600e-07	0.00
1	0.005610	0.016900	0.132231	0.000577	1.0	0.020844	0.139498	1.716100e-10	0.05
2	0.013314	0.063001	0.074380	0.005586	1.0	0.002244	0.171942	5.382400e-10	0.00
3	0.015499	0.064516	0.528926	0.003154	0.0	0.000269	0.140249	1.716100e-10	0.01
4	0.028914	0.025921	0.033058	0.000021	0.0	0.057274	0.456981	1.716100e-10	0.00

Now, we'll sum the squares of differences from all audio features to compute the similarity of all songs to *drivers license*.

```
distance_squared = songs_minus_DL_sq.sum(axis = 1)
distance_squared.head()
```

```
0    1.337163
1    1.438935
2    1.516317
3    1.004043
4    0.920316
dtype: float64
```

Now, we'll sort these distances to find the top 5 songs closest to drivers's license.

```
distances_sorted = distance_squared.sort_values()
distances_sorted.head()
```

```
2398      0.000000
81844     0.008633
4397      0.011160
130789    0.015018
```

```
143744    0.015058
dtype: float64
```

Using the indices of the top 5 distances, we will identify the top 5 songs most similar to *drivers license*:

```
spotify_data.loc[distances_sorted.index[0:6],:]
```

	artist_followers	genres	artist_name	artist_popularity	track_name
2398	1444702	pop	Olivia Rodrigo	88	drivers license
81844	2264501	pop	Jay Chou	74	
4397	25457	pop	Terence Lam	60	in Bb major
130789	176266	pop	Alan Tam	54	
143744	396326	pop & rock	Laura Branigan	64	How Am I Supposed to Live W
35627	1600562	pop	Tiziano Ferro	68	Non Me Lo So Spiegare

We can see the top 5 songs most similar to *drivers license* in the *track_name* column above. Interestingly, three of the five songs are Asian! These songs indeed sound similart to *drivers license*!

5.6 Correlation

Correlation may refer to any kind of association between two random variables. However, in this book, we will always consider correlation as the linear association between two random variables, or the Pearson's correlation coefficient. Note that correlation does not imply causalty and vice-versa.

The Pandas function corr() provides the pairwise correlation between all columns of a DataFrame, or between two Series. The function corrwith() provides the pairwise correlation of a DataFrame with another DataFrame or Series.

```
#Pairwise correlation amongst all columns
spotify_data.corr()
```

	artist_followers	artist_popularity	track_popularity	duration_ms	explicit	releas
artist_followers	1.000000	0.577861	0.197426	0.040435	0.082857	0.0985
artist_popularity	0.577861	1.000000	0.285565	-0.097996	0.092147	0.0620
track_popularity	0.197426	0.285565	1.000000	0.060474	0.193685	0.5683

	artist_followers	artist_popularity	track_popularity	duration_ms	explicit	releas
duration_ms	0.040435	-0.097996	0.060474	1.000000	-0.024226	0.0676
explicit	0.082857	0.092147	0.193685	-0.024226	1.000000	0.2156
release_year	0.098589	0.062007	0.568329	0.067665	0.215656	1.0000
danceability	-0.010120	0.038784	0.158507	-0.145779	0.138522	0.2047
energy	0.080085	0.039583	0.217342	0.075990	0.104734	0.3380
key	-0.000119	-0.011005	0.013369	0.007710	0.011818	0.0214
loudness	0.123771	0.045165	0.296350	0.078586	0.124410	0.4300
mode	0.004313	0.018758	-0.022486	-0.034818	-0.060350	-0.071
speechiness	-0.059933	0.236942	-0.056537	-0.332585	0.077268	-0.032
acousticness	-0.107475	-0.075715	-0.284433	-0.133960	-0.129363	-0.369
instrumentalness	-0.033986	-0.066679	-0.124283	0.067055	-0.039472	-0.149
liveness	0.002425	0.099678	-0.090479	-0.034631	-0.024283	-0.045
valence	-0.053317	-0.034501	-0.038859	-0.155354	-0.032549	-0.070
tempo	0.016524	-0.032036	0.058408	0.051046	0.006585	0.0793
time_signature	0.030826	-0.033423	0.071741	0.085015	0.043538	0.0894

Q: Which audio feature is the most correlated with *track_popularity*?

```
spotify_data.corrwith(spotify_data.track_popularity).sort_values(ascending = False)
```

```
track_popularity     1.000000
release_year         0.568329
loudness             0.296350
artist_popularity    0.285565
energy               0.217342
artist_followers     0.197426
explicit             0.193685
danceability         0.158507
time_signature       0.071741
duration_ms          0.060474
tempo                0.058408
key                  0.013369
mode                -0.022486
valence             -0.038859
speechiness         -0.056537
liveness            -0.090479
instrumentalness    -0.124283
acousticness        -0.284433
dtype: float64
```

Loudness is the audio feature having the highest correlation with *track_popularity*.

6 Data visualization

It is generally easier for humans to comprehend information with plots, diagrams and pictures, rather than with text and numbers. This makes data visualizations a vital part of data science. Some of the key purposes of data visualization are:

1. Data visualization is the first step towards exploratory data analysis (EDA), which reveals trends, patterns, insights, or even irrugalarities in data.
2. Data visualization can help explain the workings of complex mathematical models.
3. Data visualization are an elegant way to summarise the findings of a data analysis project.
4. Data visualizations (especially interactive ones such as those on Tableau) may be the end-product of data analytics project, where the stakeholders make decisions based on the vizualizations.

We'll use a couple of libraries for making data visualizations - matplotlib and seaborn. Matplotlib is mostly used for creating relatively simple two-dimensional plots. Its plotting interface that is similar to the `plot()` function in MATLAB, so those who have used MATLAB should find it familiar. Seaborn is a recently developed data visualization library based on matplotlib. It is more oriented towards visualizing data with Pandas DataFrame and NumPy arrays. While matplotlib may also be used to create complex plots, seaborn has some built-in themes that may make it more convenient to make complex plots. Seaborn also has color schemes and plot styles that improve the readability and aesthetics of malplotlib plots. However, preferences depend on the user and their coding style, and it is perfectly fine to use either library for making the same visualization.

Let's visualize the life expectancy of different countries with GDP per capita. We'll read the data file *gdp_lifeExpectancy.csv*, which contains the GDP per capita and life expectancy of countries from 1952 to 2007.

```
import pandas as pd
import numpy as np
```

```
gdp_data = pd.read_csv('./Datasets/gdp_lifeExpectancy.csv')
gdp_data.head()
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

6.0.1 Scatterplots and trendline

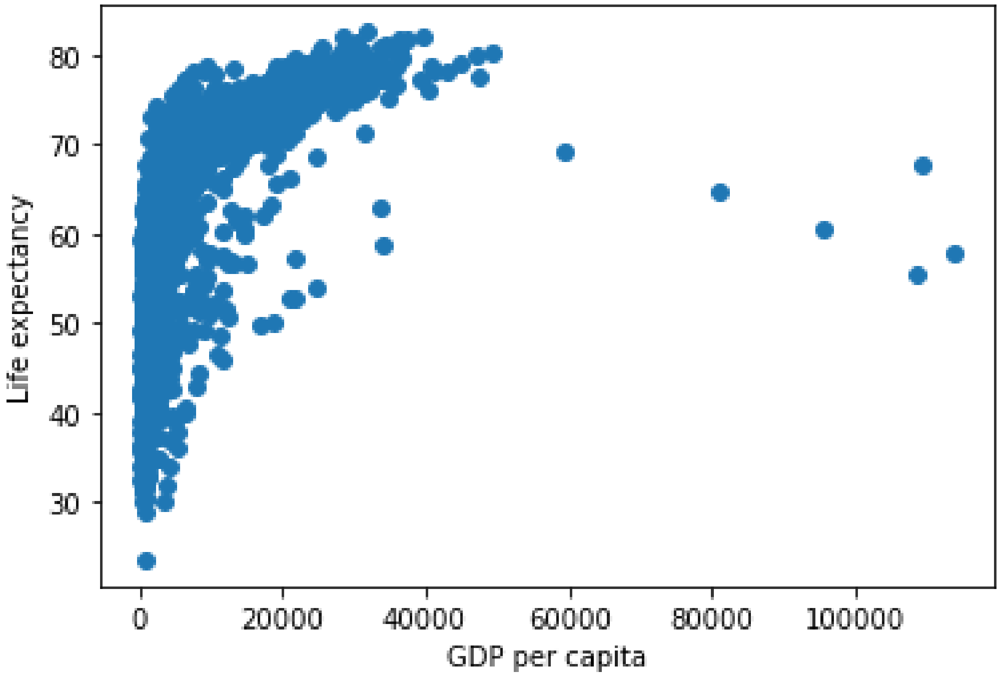
Now, we'll import the pyplot module of matplotlib to make plots. We'll use the plot() function to make the scatter plot, and the functions xlabel() and ylabel() for lablelling the plot axes.

```
import matplotlib.pyplot as plt
```

Q: Make a scatterplot of Life expectancy vs GDP per capita.

```
#Making a scatterplot of Life expectancy vs GDP per capita
x = gdp_data.gdpPercap
y = gdp_data.lifeExp
plt.plot(x,y,'o') #By default, the plot() function makes a lineplot. The 'o' arguments spe
plt.xlabel('GDP per capita')  #Labelling the horizontal X-axis
plt.ylabel('Life expectancy') #Labelling the verical Y-axis
```

```
Text(0, 0.5, 'Life expectancy')
```



From the above plot, we observe that life expectancy seems to be positively correlated with the GDP per capita of the country, as one may expect. However, there are a few outliers in the data - which are countries having extremely high GDP per capita, but not a correspondingly high life expectancy.

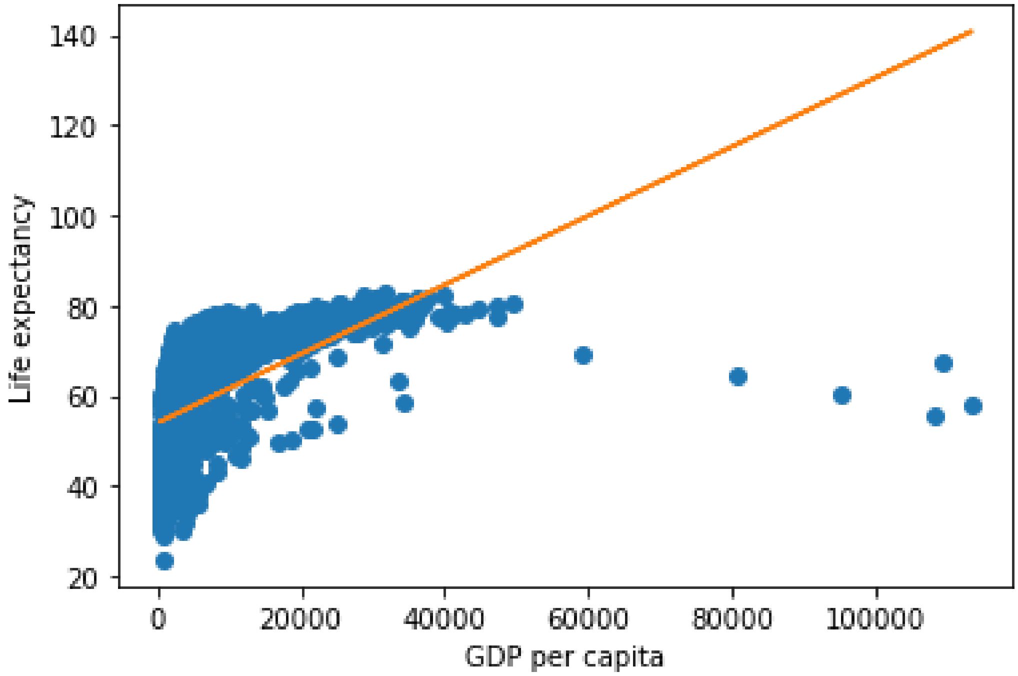
Sometimes it is difficult to get an idea of the overall trend (positive or negative correlation). In such cases, it may help to add a trendline to the scatter plot. In the plot below we add a trendline over the scatterplot showing that the life expectancy on an average increases with increasing GDP per capita. The trendline is actually a linear regression of life expectancy on GDP per capita. However, we'll not discuss linear regression is this book.

Q: Add a trendline over the scatterplot of life expectancy vs GDP per capita.

```
#Making a scatterplot of Life expectancy vs GDP per capita
x = gdp_data.gdpPercap
y = gdp_data.lifeExp
plt.plot(x,y,'o') #By default, the plot() function makes a lineplot. The 'o' arguments spe
plt.xlabel('GDP per capita')  #Labelling the horizontal X-axis
plt.ylabel('Life expectancy') #Labelling the verical Y-axis

#Plotting a trendline (linear regression) on the scatterplot
slope_intercept_trendline = np.polyfit(x,y,1)   #Finding the slope and intercept for the t
```

```
compute_y_given_x = np.poly1d(slope_intercept_trendline) #Defining a function that compute
plt.plot(x,compute_y_given_x(x)) #Plotting the trendline
```



The above plot shows that our earlier intuition of a postive correlation between Life expectancy and GDP per capita was correct.

We used the NumPy function polyfit() to compute the slope and intercept of the trendline. Then, we defined an object `compute_y_given_x` of poly1d class and used it to compute the trendline.

6.0.2 Subplots

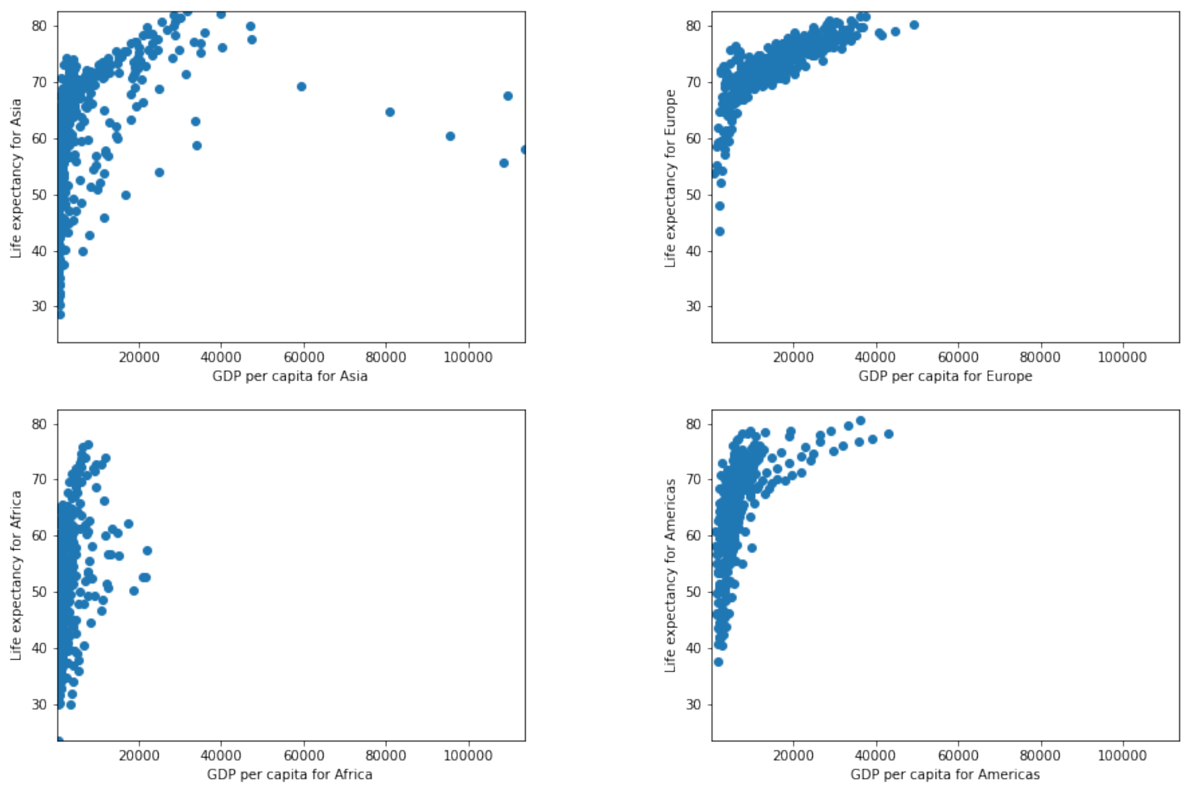
There is often a need to make a few plots together to compare them. See the example below.

Q: Make scatterplots of life expectancy vs GDP per capita separately for each of the 4 continents of Asia, Europe, Africa and America. Arrange the plots in a 2 x 2 grid.

```
#Defining a 2x2 grid of subplots
fig, axes = plt.subplots(2,2,figsize=(15,10))
plt.subplots_adjust(wspace=0.4) #adjusting white space between individual plots
```

```python
#Making a scatterplot of Life expectancy vs GDP per capita for each continent
continents = np.array([['Asia', 'Europe'], ['Africa', 'Americas']])

#Looping over the 2x2 grid
for i in range(2):
    for j in range(2):
        x = gdp_data.loc[gdp_data.continent==continents[i,j],:].gdpPercap
        y = gdp_data.loc[gdp_data.continent==continents[i,j],:].lifeExp
        axes[i,j].plot(x,y,'o')
        axes[i,j].set_xlim([gdp_data.gdpPercap.min(), gdp_data.gdpPercap.max()])
        axes[i,j].set_ylim([gdp_data.lifeExp.min(), gdp_data.lifeExp.max()])
        axes[i,j].set_xlabel('GDP per capita for '+ continents[i,j])
        axes[i,j].set_ylabel('Life expectancy for '+ continents[i,j])
```



We observe that for each continent, except Africa, initially life expectancy increases rapidly with increasing GDP per capita. However, after a certain threshold of GDP per capita, life expectancy increases slowly. Several countries in Europe enjoy a relatively high GDP per capita as well as high life expectancy. Some countries in Asia have an extremely high GDP per capita, but a relatively low life expectancy. It will be interesting to see the proportion of GDP associated with healthcare for these outlying Asian countries, and European countries.

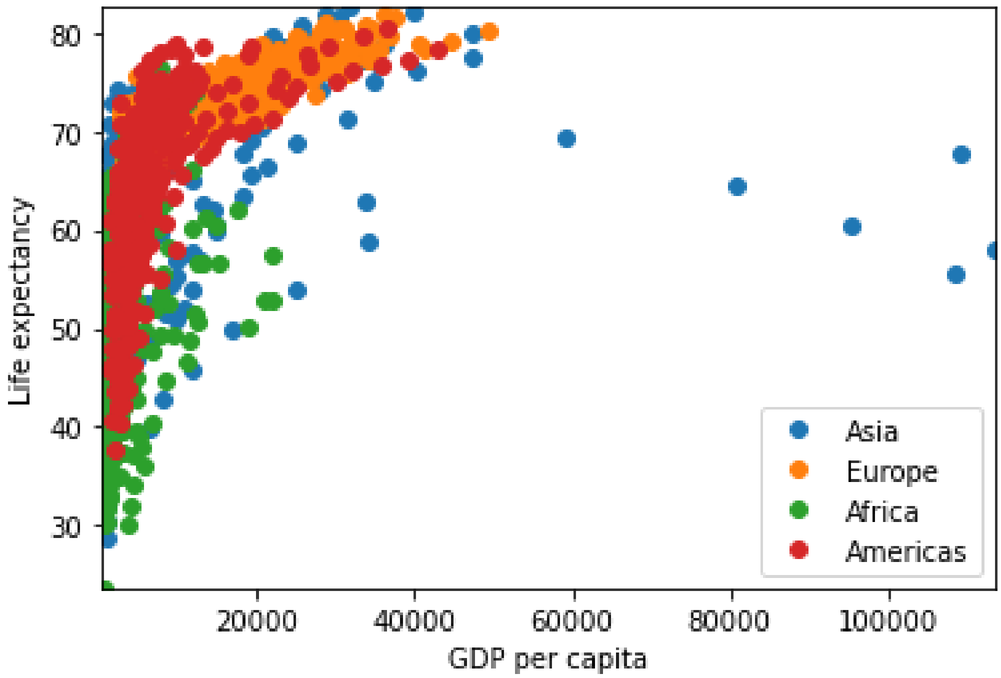
We used the subplot function of matplotlib to define the 2x2 grid of subplots. The function subplots_adjust() can be used to adjust white spaces around the plot. We used a `for` loop to iterate over each subplot. The `axes` object returned by the `subplot()` function was used to refer to individual subplots.

6.0.3 Overlapping plots with legend

We can also have the scatterplot of all the continents on the sample plot, with a distinct color for each continent. A legend will be required to identify the continent's color.

```
continents = np.array([['Asia', 'Europe'], ['Africa', 'Americas']])
for i in range(2):
    for j in range(2):
        x = gdp_data.loc[gdp_data.continent==continents[i,j],:].gdpPercap
        y = gdp_data.loc[gdp_data.continent==continents[i,j],:].lifeExp
        plt.plot(x,y,'o',label = continents[i,j])
        plt.xlim([gdp_data.gdpPercap.min(), gdp_data.gdpPercap.max()])
        plt.ylim([gdp_data.lifeExp.min(), gdp_data.lifeExp.max()])
        plt.xlabel('GDP per capita')
        plt.ylabel('Life expectancy')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1d09bf00040>
```



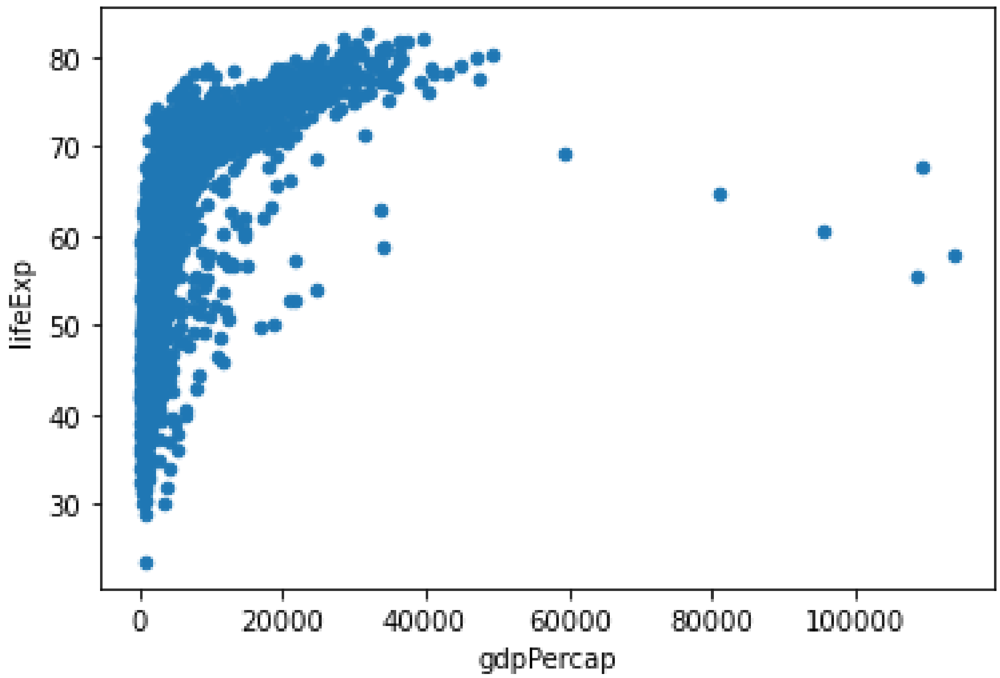
6.1 Pandas

Matplotlib is low-level tool, in which different components of the plot, such as points, legend, axis titles, etc. need to be specified separately. The Pandas plot() function can be used directly with a DataFrame or Series to make plots.

6.1.1 Scatterplots and lineplots

```
#Plotting life expectancy vs GDP per capita using the Pandas plot() function
gdp_data.plot(x = 'gdpPercap', y = 'lifeExp', kind = 'scatter')
```

```
<AxesSubplot:xlabel='gdpPercap', ylabel='lifeExp'>
```



Note that with matplolib, it will take 3 lines to make the same plot - one for the scatterplot, and two for the axis titles.

Let us re-arrange the data to show other benefits of the Pandas `plot()` function. Note that data resphaping is explained in Chapter 8 of the book, so you may ignore the code block below that uses the pivot_table() function.

```
#You may ignore this code block until Chapter 8.
mean_gdp_per_capita = gdp_data.pivot_table(index = 'year', columns = 'continent',values =
mean_gdp_per_capita.head()
```

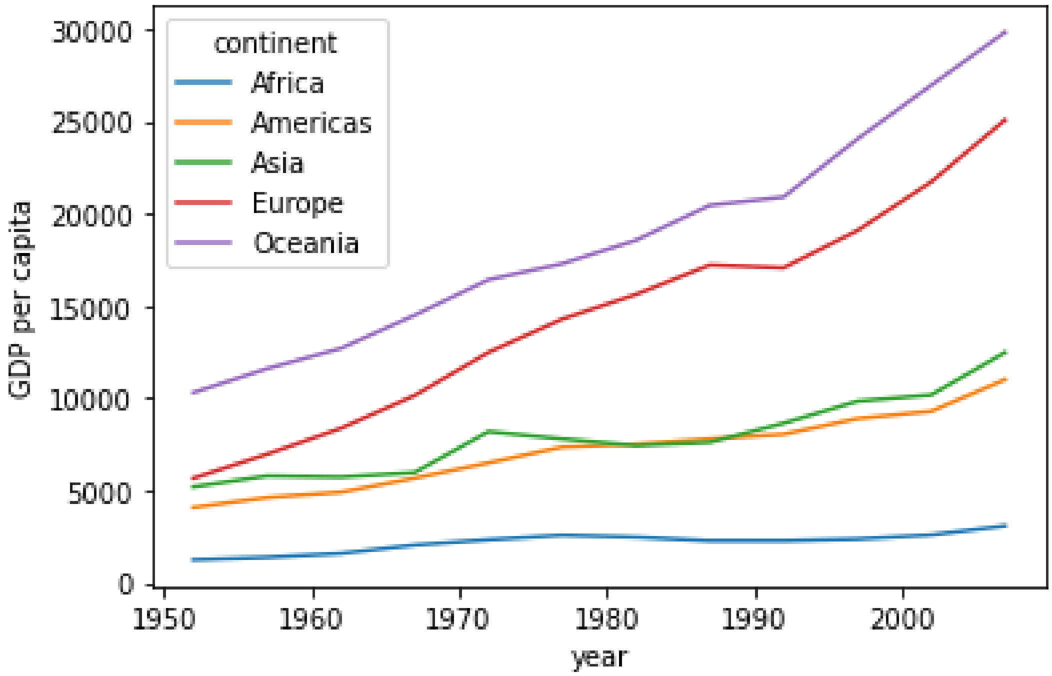
continent year	Africa	Americas	Asia	Europe	Oceania
1952	1252.572466	4079.062552	5195.484004	5661.057435	10298.085650
1957	1385.236062	4616.043733	5787.732940	6963.012816	11598.522455
1962	1598.078825	4901.541870	5729.369625	8365.486814	12696.452430
1967	2050.363801	5668.253496	5971.173374	10143.823757	14495.021790
1972	2339.615674	6491.334139	8187.468699	12479.575246	16417.333380

We have reshaped the data to obtain the mean GDP per capita of each continent for each year.

The pandas `plot()` function can be directly used with this DataFrame to create line plots showing mean GDP per capita of each continent with year.

```
mean_gdp_per_capita.plot(ylabel = 'GDP per capita')
```

```
<AxesSubplot:xlabel='year', ylabel='GDP per capita'>
```



We observe that the mean GDP per capita of of Europe and Oceania have increased rapidly, while that for Africa is incresing very slowly.

The above plot will take several lines of code if developed using only matplotlib. The pandas `plot()` function has a framework to conveniently make commonly used plots.

6.1.2 Bar plots

Bar plots can be made using the pandas bar function with the DataFrame or Series, just like the line plots and scatterplots.

Below, we are reading the dataset of noise complaints of type *Loud music/Party* received the police in New York City in 2016.

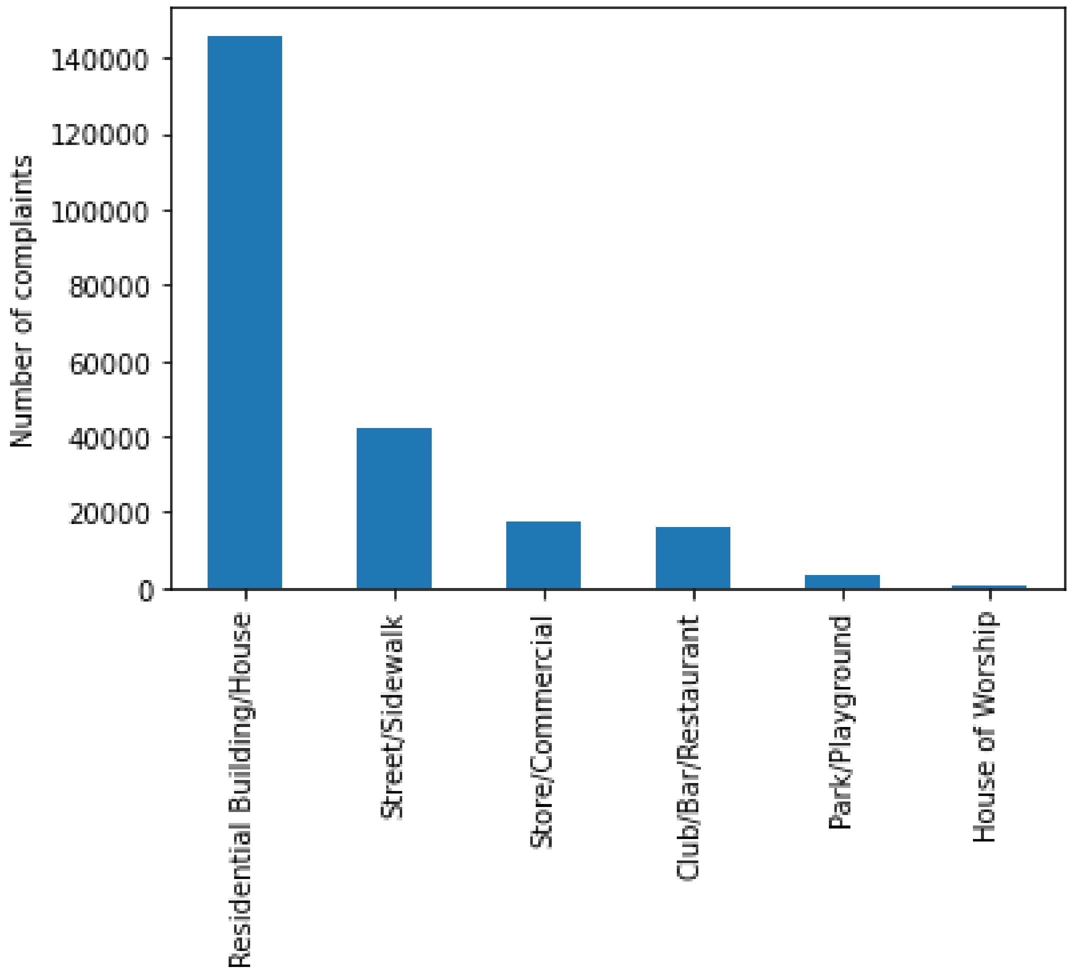
```
nyc_party_complaints = pd.read_csv('./Datasets/party_nyc.csv')
nyc_party_complaints.head()
```

	Created Date	Closed Date	Location Type	Incident Zip	City	Borough
0	12/31/2015 0:01	12/31/2015 3:48	Store/Commercial	10034.0	NEW YORK	MANHA
1	12/31/2015 0:02	12/31/2015 4:36	Store/Commercial	10040.0	NEW YORK	MANHA
2	12/31/2015 0:03	12/31/2015 0:40	Residential Building/House	10026.0	NEW YORK	MANHA
3	12/31/2015 0:03	12/31/2015 1:53	Residential Building/House	11231.0	BROOKLYN	BROOK
4	12/31/2015 0:05	12/31/2015 3:49	Residential Building/House	10033.0	NEW YORK	MANHA

Let us visualise the locations from where the the complaints are coming.

```
#Using the pandas function bar() to create bar plot
nyc_party_complaints['Location Type'].value_counts().plot.bar(ylabel = 'Number of complain
```

```
<AxesSubplot:ylabel='Number of complaints'>
```

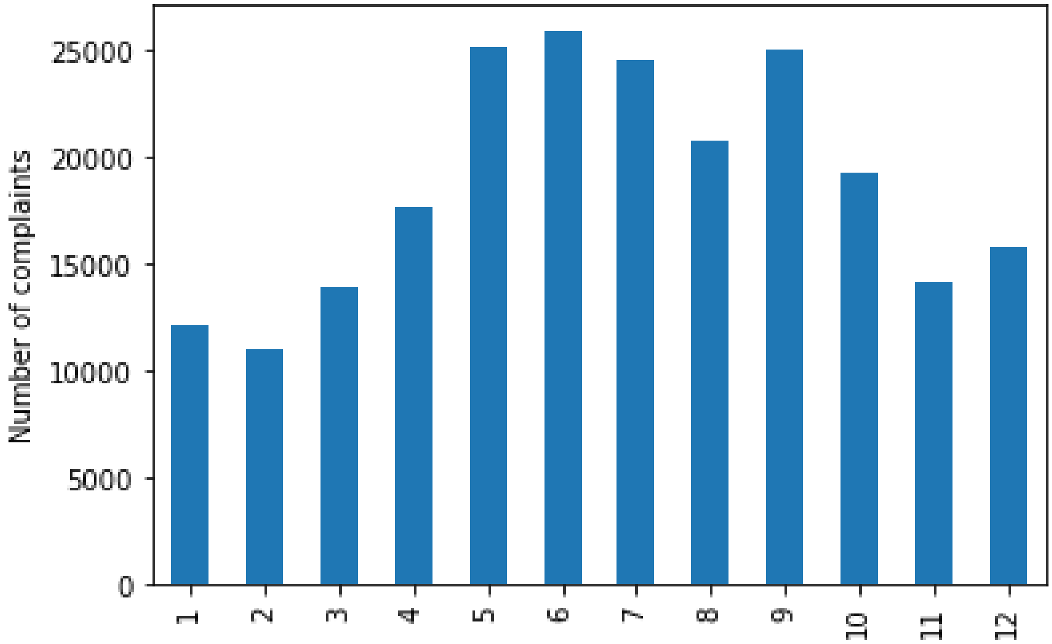


From the above plot, we observe that most of the complaints come from residential buildings and houses, as one may expect.

Let is visualize the time of the year when most complaints occur.

```
#Using the pandas function bar() to create bar plot
nyc_party_complaints['Month_of_the_year'].value_counts().sort_index().plot.bar(ylabel = 'N
```

```
<AxesSubplot:ylabel='Number of complaints'>
```



Try executing the code without sort_index() to figure out the purpose of using the function.

From the above plot, we observe that most of the complaints occur during summer and early Fall.

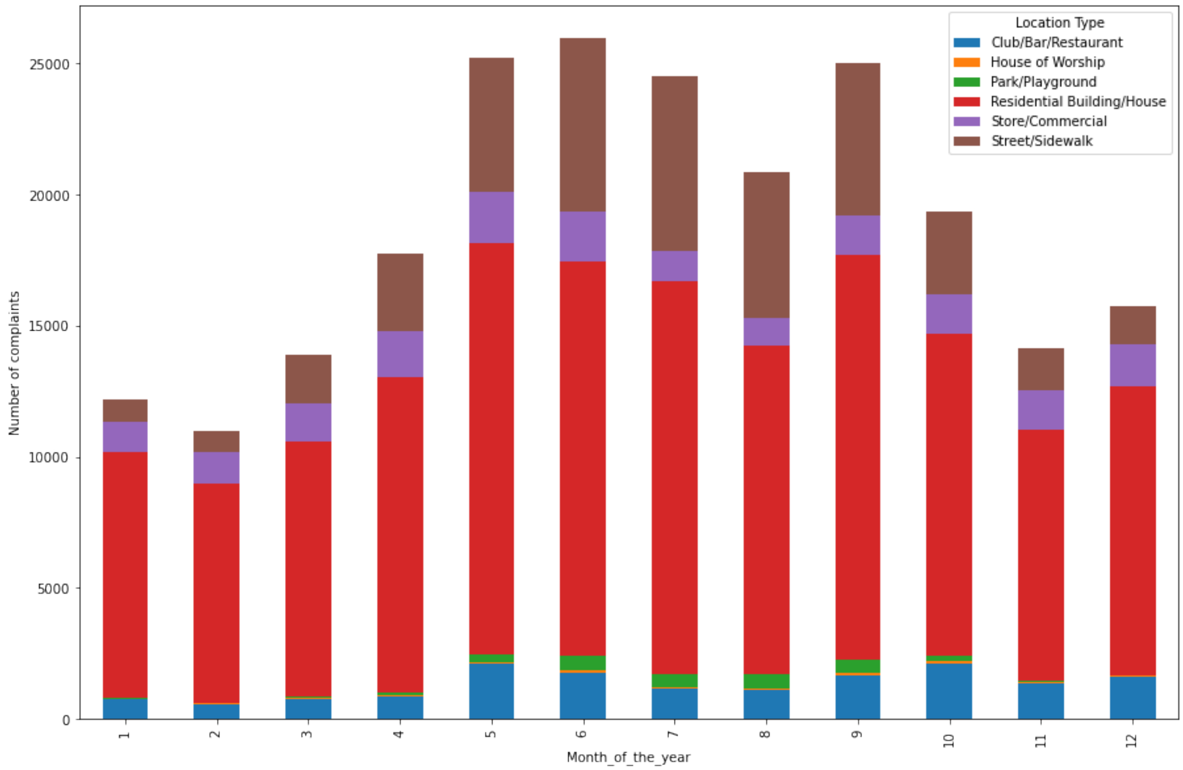
Let us create a stacked bar chart that combines both the above plots into a single plot. You may ignore the code used for re-shaping the data until Chapter 8. The purpose here is to show the utility of the pandas `bar()` function.

```
#Reshaping the data to make it suitable for a stacked barplot - ignore this code until cha
complaints_location=pd.crosstab(nyc_party_complaints.Month_of_the_year, nyc_party_complain
complaints_location.head()
```

Location Type Month_of_the_year	Club/Bar/Restaurant	House of Worship	Park/Playground	Residential Building/H
1	748	24	17	9393
2	570	29	16	8383
3	747	39	90	9689
4	848	53	129	11984
5	2091	72	322	15676

```
#Stacked bar plot showing number of complaints at different months of the year, and from d
complaints_location.plot.bar(stacked=True,ylabel = 'Number of complaints',figsize=(15, 10)
```

```
<AxesSubplot:xlabel='Month_of_the_year', ylabel='Number of complaints'>
```



The above plots gives the insights about location and day of the year simultaneously that were previously separately obtained by the individual plots.

6.2 Seaborn

Seaborn offers the flexibility of simultaneously visualizing multiple variables in a single plot, and offers several themes to develop plots.

```
#Importing the seaborn library
import seaborn as sns
```

6.2.1 Bar plots with confidence interevals

We'll group the data to obtain the total complaints for each *Location Type, Borough, Month_of_the_year, and Hour_of_the_day.* Note that you'll learn grouping data in Chapter 9, so you may ignore the next code block. The grouping is done to shape the data in a suitable form for visualization.

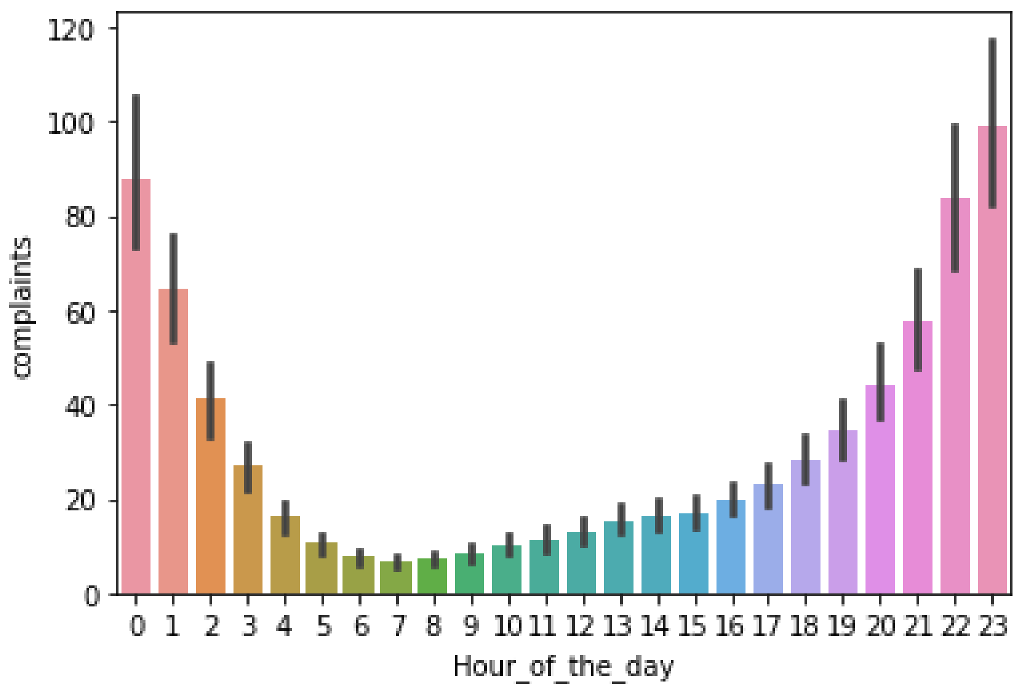
```
#Grouping the data to make it suitable for visualization using Seaborn. Ignore this code b
nyc_complaints_grouped = nyc_party_complaints[['Location Type','Borough','Month_of_the_yea
nyc_complaints_grouped.head()
```

	Location Type	Borough	Month_of_the_year	Hour_of_the_day	complaints
0	Club/Bar/Restaurant	BRONX	1	0	10
1	Club/Bar/Restaurant	BRONX	1	1	10
2	Club/Bar/Restaurant	BRONX	1	2	6
3	Club/Bar/Restaurant	BRONX	1	3	6
4	Club/Bar/Restaurant	BRONX	1	4	3

Let us create a bar plot visualizing the average number of complaints with the time of the day.

```
sns.barplot(x="Hour_of_the_day", y = 'complaints',  data=nyc_complaints_grouped)
```

```
<AxesSubplot:xlabel='Hour_of_the_day', ylabel='complaints'>
```



From the above plot, we observe that most of the complaints are made around midnight. However, interestingly, there are some complaints at each hour of the day.

Note that the above barplot shows the mean number of complaints in a month at each hour of the day. The black lines are the 95% confidence intervals of the mean number of complaints.

6.2.2 Facetgrid: Multi-plot grid for plotting conditional relationships

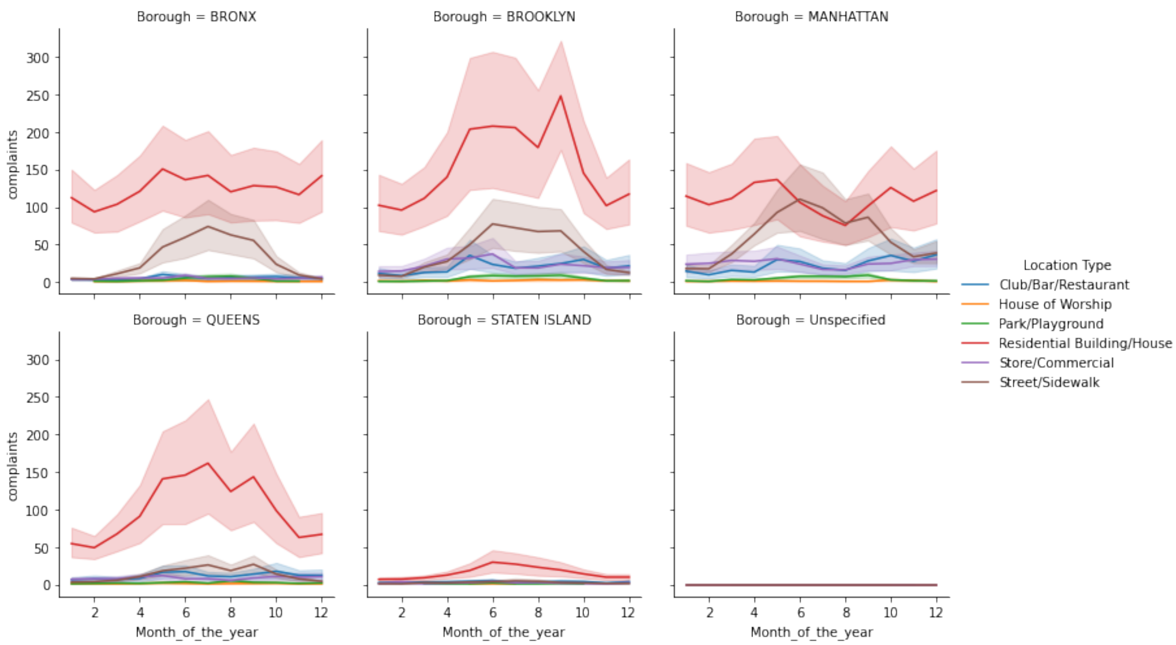
With pandas, we simultaneously visualized the number of complaints with month of the year and location type. We'll use Seaborn to add another variable - Borough to the visualization.

Q: Visualize the number of complaints with *Month_of_the_year, Location Type,* and *Borough.*

The seaborn class FacetGrid is used to design the plot, i.e., specify the way the data will be divided in mutually exclusive subsets for visualization. Then the [map] function of the `FacetGrid` class is used to apply a plotting function to each subset of the data.

```
#Visualizing the number of complaints with Month_of_the_year, Location Type, and Borough.
a = sns.FacetGrid(nyc_complaints_grouped, hue = 'Location Type', col = 'Borough',col_wrap=
a.map(sns.lineplot,'Month_of_the_year','complaints')
a.add_legend()
```

```
<seaborn.axisgrid.FacetGrid at 0x1d0e52ff580>
```



From the above plot, we get a couple of interesting insights: 1. For Queens and Staten Island, most of the complaints occur in summer, for Manhattan and Bronx it is mostly during late spring, while Brooklyn has a spike of complaints in early Fall. 2. In most of the Boroughs, the majority complaints always occur in residential areas. However, for Manhattan, the number of street/sidewalk complaints in the summer are comparable to those from residential areas.

We have visualized 4 variables simultaneously in the above plot.

Let us consider another example, where we will visualize the weather in a few cities of Australia. The file *Australia_weather.csv* consists of weather details of Syndey, Canberra, and Melbourne from 2007 to 2017.

```
aussie_weather = pd.read_csv('./Datasets/Australia_weather.csv')
aussie_weather.head()
```

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	Win
0	10/20/2010	Sydney	12.9	20.3	0.2	3.0	10.9	ENE	37
1	10/21/2010	Sydney	13.3	21.5	0.0	6.6	11.0	ENE	41
2	10/22/2010	Sydney	15.3	23.0	0.0	5.6	11.0	NNE	41
3	10/26/2010	Sydney	12.9	26.7	0.2	3.8	12.1	NE	33

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	Win
4	10/27/2010	Sydney	14.8	23.8	0.0	6.8	9.6	SSE	54

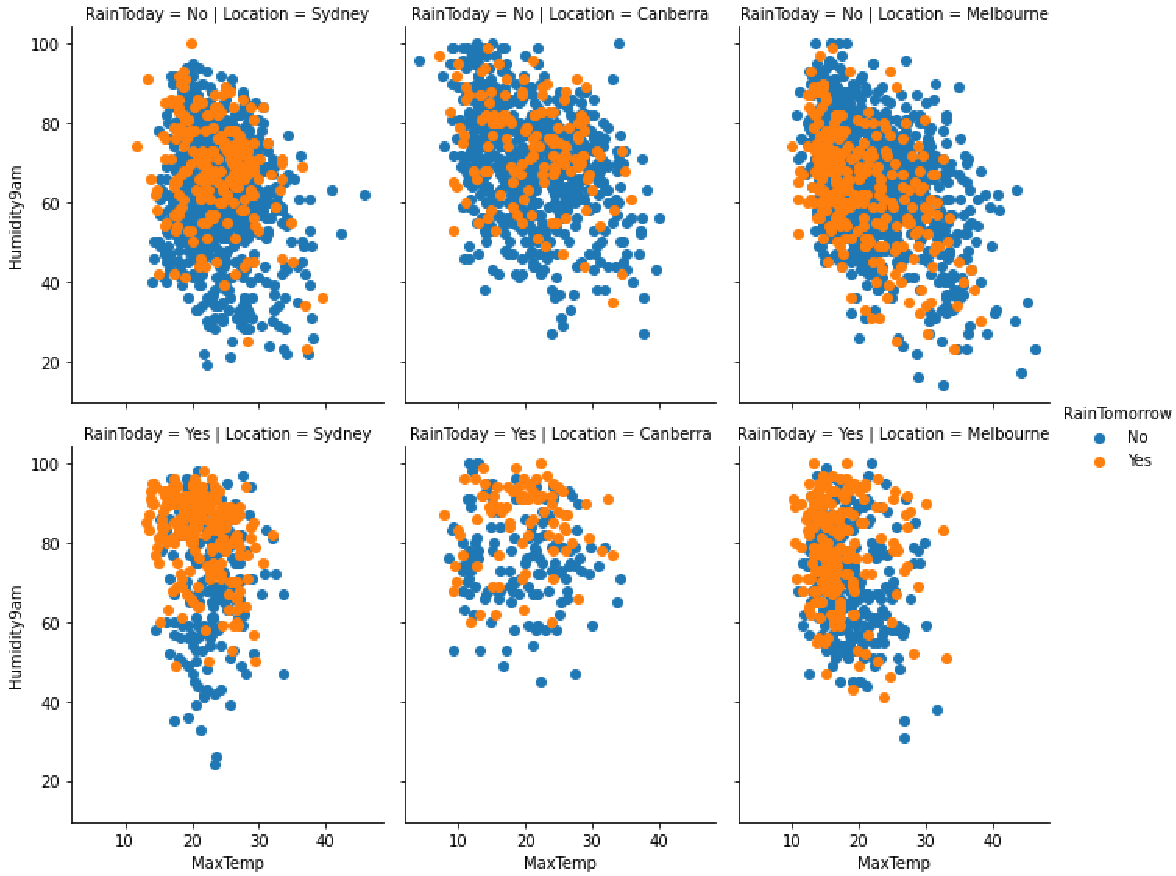
```
aussie_weather.shape
```

```
(4666, 24)
```

Q: Visualize if it rains the next day (*RainTomorrow*) given whether it has rained today (*RainToday*), the current day's humidity (*Humidity9am*), maximum temperature (*MaxTemp*) and the city (*Location*).

```
a = sns.FacetGrid(aussie_weather,col='Location',row='RainToday',height = 4,aspect = 0.8,hu
a.map(plt.scatter,'MaxTemp','Humidity9am')
a.add_legend()
```

```
<seaborn.axisgrid.FacetGrid at 0x1d0e77b0610>
```



Humidity tends to be higher when it is going to rain the next day. However, the correlation is much more pronounced for Syndey. In case it is not raining on the current day, humidity seems to be slightly negatively correlated with temperature.

6.2.3 Histogram and density plots

Histogram and density plots visualize the data distribution. A histogram plots the number of observations occuring within discrete, evenly spaced bins of a random variable, to visualize the distribution of the variable. It may be considered a special case of bar plot as bars are used to plot the observation counts.

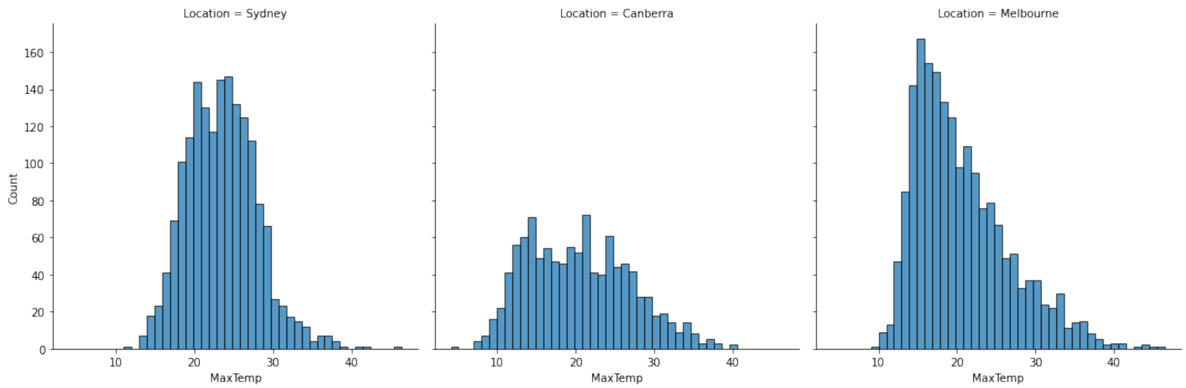
A density plot uses a kernel density estimate to approximate the distribution of random variable.

We can use the Seaborn displot() function to make both kinds of plots - histogram or density plot.

Example: Make a histogram showing the distributions of maximum temperature in Sydney, Canberra and Melbourne.

```
sns.displot(data = aussie_weather, x = 'MaxTemp',kind = 'hist',col='Location')
```

<seaborn.axisgrid.FacetGrid at 0x1d0ec989e50>

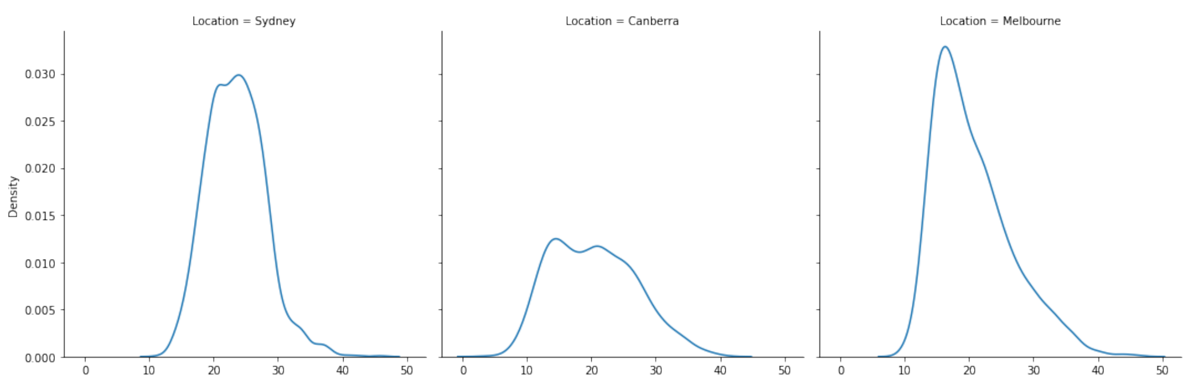


From the above plot, we observe that: 1. Melbourne has a right skewed distribution with the median temperature being smaller than the mean. 2. Canberra seems to have the highest variation in the temperture.

Example: Make a density plot showing the distributions of maximum temperature in Sydney, Canberra and Melbourne.

```
sns.displot(data = aussie_weather, x = 'MaxTemp',kind = 'kde', col = 'Location')
```

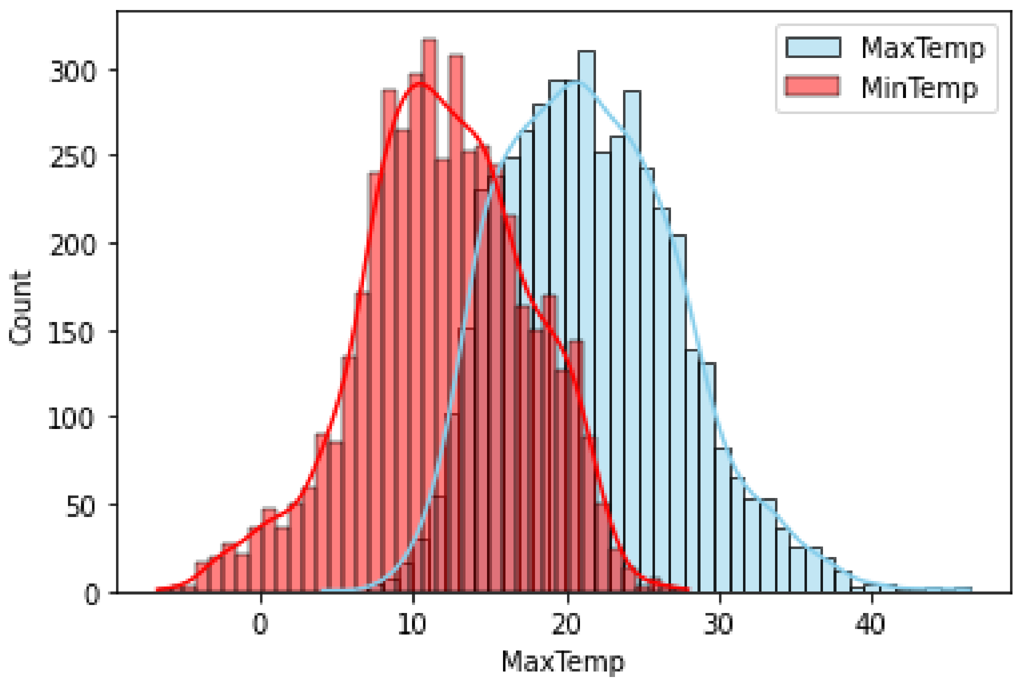
<seaborn.axisgrid.FacetGrid at 0x1d0e963f8e0>



Example: Show the distributions of the maximum and minimum temperatures in a single plot.

```
sns.histplot(data=aussie_weather, x="MaxTemp", color="skyblue", label="MaxTemp", kde=True)
sns.histplot(data=aussie_weather, x="MinTemp", color="red", label="MinTemp", kde=True)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1d0eb6b9790>
```



The Seaborn function histplot() can be used to make a density plot overlapping on a histogram.

6.2.4 Boxplots

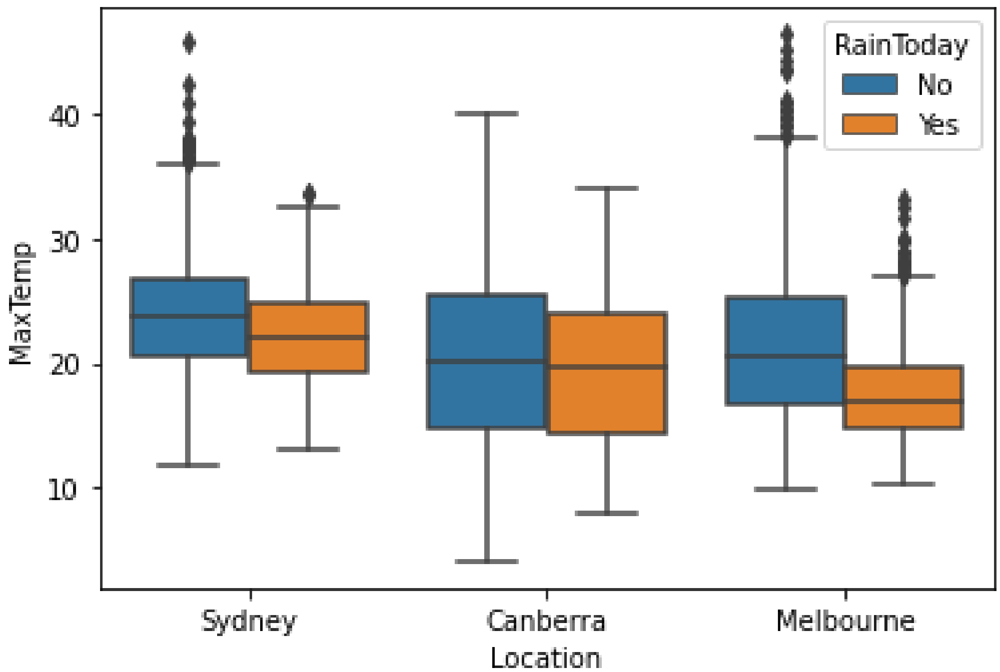
Boxplots is a standardized way of visualizing the data distribution. They show five key metrics that describe the data distribution - median, 25th percentile value, 75th percentile value, minimum and maximum, as shown in the figure below. Note that the minimum and maximum exclude the outliers.

```
<IPython.core.display.Image object>
```

Example: Make a boxplot comparing the distributions of maximum temperatures of Sydney, Canberra and Melbourne, given whether or not it has rained on the day.

```
sns.boxplot(data = aussie_weather,x = 'Location', y = 'MaxTemp',hue = 'RainToday')
```

```
<AxesSubplot:xlabel='Location', ylabel='MaxTemp'>
```



From the above plot, we observe that: 1. The maximum temperature of the day, on an average, is lower if it rained on the day. 2. Sydney and Meloboure have some extremly high outlying values of maximum temperature.

We have used the Seaborn boxplot() function for the above plot.

7 Data cleaning and preparation

Missing values in a dataset can occur due to several reasons such as breakdown of measuring equipment, accidental removal of observations, lack of response by respondents, error on the part of the reearcher, etc.

Removing all rows / columns with even a single missing value results in loss of data that is non-missing in the respective rows/columns. In this section, we'll see how to make a smart guess for the missing values, so that we can (hopefully) maximize the information that can be extracted from the data.

7.0.1 Types of missing values

Rubin (1976) classified missing values into three categories, depending on how the values could have been missing.

7.0.2 Missing Completely at Random (MCAR)

If the probablity that a random variable's value is missing is the same for all the observations, then the data is said to be missing completely at random. An example of MCAR is a weighing scale that ran out of batteries. Some of the data will be missing simply because of bad luck.

7.0.3 Missing at Random (MAR)

If the probability of being missing is the same only within groups defined by the observed data, then the data are missing at random (MAR). MAR is a much broader class than MCAR. For example, when placed on a soft surface, a weighing scale may produce more missing values than when placed on a hard surface. Such data are thus not MCAR. If, however, we know surface type and if we can assume MCAR within the type of surface, then the data are MAR

7.0.4 Missing Not at Random (MNAR)

MNAR means that the probability of being missing varies for reasons that are unknown to us. For example, the weighing scale mechanism may wear out over time, producing more missing data as time progresses, but we may fail to note this. If the heavier objects are measured later in time, then we obtain a distribution of the measurements that will be distorted. MNAR includes the possibility that the scale produces more missing values for the heavier objects (as above), a situation that might be difficult to recognize and handle.

Source: https://stefvanbuuren.name/fimd/sec-MCAR.html

8 Data wrangling

Data wrangling

9 Data aggregation

Data aggregation

10 Datasets

Datasets used in the book can be found here

References

Rubin, Donald B. 1976. “Inference and Missing Data.” *Biometrika* 63 (3): 581–92.