Introduction to Data Science with Python

Arvind Krishna, Lizhen Shi, Emre Besler, and Arend Kuyper

2022-09-20

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. Students may use the STAT201 book (https://nustat.github.io/Intro_to_programming_for_data_sci/) to review the python basics required for the STAT303 sequence. The core part of the course begins from the third chapter - *Reading data*.

Please feel free to let the instructors know in case of any typos/mistakes/general feedback in this book.

Part I

Quick overview: Python programming

1 Introduction to Python and Jupyter Notebooks

This chapter is a very brief introduction to python and Jupyter notebooks. If you have not taken STAT201 (Introduction to programming for data science), which is now a pre-requisite for the data science major / minor program, please review the python programming section (chapters 1-6) from the STAT201 book. It is assumed that you are already comfortable with this content. Some of the content of these chapters is reviewed briefly in the first two chapters of this book.

1.1 Installation

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.

1.2 Jupyter notebook

1.2.1 Introduction

Jupyter notebook is an interactive platform, where you can write code and text, and make visualizations. Jupyter is a loose acronym meaning Julia, Python, and R. However, now it supports Ruby, Haskell, Scala, Go, etc, besides Julia, Python, and R. 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.

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

1.2.2 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 the notebook by clicking on the text, which says 'Untitled'.

1.2.3 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 extension *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.

1.2.4 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>
```

1.3 Python language basics

1.3.1 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.

1.3.2 Assigning variable name to object

1.3.2.1 Call by reference

Python utilizes a system, which is known as *Call by Object Reference*. 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.

1.3.2.2 Assigning multiple variable names

Values can be assigned to multiple variables in a single statement by separating the variable names and values with commas.

```
color1, color2, color3 = "red", "green", "blue"
```

```
color1
```

```
'red'
```

```
color3
```

```
'blue'
```

The same value can be assigned to multiple variables by chaining multiple assignment operations within a single statement.

```
color4 = color5 = color6 = "magenta"
```

1.3.2.3 Rules for variable names

Variable names can be short (`a`, `x`, `y`, etc.) or descriptive (`my_favorite_color`, `profit_margin`, `the_3_musketeers`, etc.). However, we recommend that you use descriptive variable names as it makes it easier to understand the code.

The rules below must be followed while naming Python variables:

- A variable's name must start with a letter or the underscore character `_`. It cannot begin with a number.
- A variable name can only contain lowercase (small) or uppercase (capital) letters, digits, or underscores (`a-z`, `A-Z`, `0-9`, and `_`).
- Variable names are case-sensitive, i.e., `a_variable`, `A_Variable`, and `A_VARIABLE` are all different variables.

Here are some valid variable names:

```
a_variable = 23
is_today_Saturday = False
my_favorite_car = "Delorean"
the_3_musketeers = ["Athos", "Porthos", "Aramis"]
```

Let's try creating some variables with invalid names. Python prints a syntax error if the variable's name is invalid.

> **Syntax**: The syntax of a programming language refers to the rules that govern the structure of a valid instruction or *statement*. If a statement does not follow these rules, Python stops execution and informs you that there is a *syntax error*. Syntax can be thought of as the rules of grammar for a programming language.

```
a variable = 23
```

```
is_today_$aturday = False
```

```
my-favorite-car = "Delorean"
```

```
3_musketeers = ["Athos", "Porthos", "Aramis"]
```

1.3.3 Built-in objects

1.3.3.1 Built-in data types

Variable is created as soon as a value is assigned to it. We don't have to define the type of variable explicitly as in other programming languages because Python can automatically guess the type of data entered **(dynamically typed)**.

Any data or information stored within a Python variable has a *type*. We can view the type of data stored within a variable using the `type` function.

```
a_variable
```

```
23
```

```
type(a_variable)
```

```
int
```

```
is_today_Saturday
```

```
False
```

```
type(is_today_Saturday)
```

```
bool
```

```
my_favorite_car
```

```
'Delorean'
```

```
type(my_favorite_car)
```

```
str
```

```python
the_3_musketeers
```

```
['Athos', 'Porthos', 'Aramis']
```

```python
type(the_3_musketeers)
```

```
list
```

Python has several built-in data types for storing different kinds of information in variables.

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

Primitive: Integer, float, boolean, None, and string are *primitive data types* because they represent a single value.

Containers: Other data types like list, tuple, and dictionary are often called *data structures* or *containers* because they hold multiple pieces of data together. We'll discuss these datatypes in chapter 2.

The data type of the object can be identified using the in-built python function `type()`. For example, see the following objects and their types:

```python
type(4)
```

```
int
```

```python
type(4.4)
```

```
float
```

```python
type('4')
```

```
str
```

```python
type(True)
```

```
bool
```

1.3.3.2 Built-in modules and functions

Built-in functions in Python are a set of predefined functions that are available for use without the need to import any additional libraries or modules. The Python Standard Library is very extensive. Besides built-in functions, it also contains many Python scripts (with the . py extension) containing useful utilities and modules written in Python that provide standardized solutions for many problems that occur in everyday programming.

Below are a couple of examples:

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]
```

The advantage of the range type over a regular list or tuple is that a range object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the start, stop and step values, calculating individual items and subranges as needed).

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'
```

1.3.4 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**: NumPy is a fundamental library for numerical computing in Python. It provides support for arrays, matrices, and mathematical functions, making it essential for scientific and data analysis tasks.. It is mostly used for performing numerical operations and efficiently storing numerical data.

2. **Pandas**: Pandas is a powerful data manipulation and analysis library. It offers data structures like DataFrames and Series, which facilitate data reading, cleaning, transformation, and analysis, making it indispensable in data science projects.

3. **Matplotlib, Seaborn**: Matplotlib is a comprehensive library for creating static, animated, or interactive plots and visualizations. It is commonly used for data visualization and exploration in data science. Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

4. **SciPy**: SciPy is used for performing scientific computing such as solving differential equations, optimization, statistical tests, etc.

5. **Scikit-learn**: Scikit-learn is a machine learning library that provides a wide range of tools for data pre-procesing, classification, regression, clustering, dimensionality reduction, and more. It simplifies the implementation of machine learning algorithms and model evaluation.

6. **Statsmodels**: Statsmodels is used for developing statistical models with a focus on inference *(in contrast to focus on prediction as in scikit-learn).*

To use libraries like NumPy, pandas, Matplotlib, and scikit-learn in Python, you typically need to follow these steps:

1. Install the libraries (Anaconda already does this)
2. Import the libraries in the python script or jupyter notebook
3. Use the Library Functions and Classes: After importing the libraries, you can use their functions, classes, and methods in your code. For instance, you can create NumPy arrays, manipulate data with pandas, create plots with Matplotlib, or train machine learning models with scikit-learn.

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])
```

There's different ways to import:

- Import the whole module using its original name: `import math, os`
- Import specific things from the module: `from random import randint from math import pi`
- Import the whole library and rename it, usually using a shorter variable name: `import pandas as pd`
- Import a specific method from the module and rename it as it is imported: `from os.path import join as join_path`

1.3.5 User-defined functions

A function is a reusable set of instructions that takes one or more inputs, performs some operations, and often returns an output. **Indeed, while python's standard library and ecosystem libraries offer a wealth of pre-defined functions for a wide range of tasks, there are situations where defining your own functions is not just beneficial but necessary.**

1.3.5.1 Creating and using functions

You can define a new function using the def keyword.

```
def say_hello():
    print('Hello there!')
    print('How are you?')
```

Note the round brackets or parentheses `()` and colon `:` after the function's name. Both are essential parts of the syntax. The function's *body* contains an indented block of statements. The statements inside a function's body are not executed when the function is defined. To execute the statements, we need to *call* or *invoke* the function.

```
say_hello()
```

```
Hello there!
How are you?
```

```
def say_hello_to(name):
    print('Hello ', name)
    print('How are you?')
```

```
say_hello_to('Lizhen')
```

```
Hello  Lizhen
How are you?
```

```
name = input ('Please enter your name: ')
say_hello_to(name)
```

```
Please enter your name: George
Hello  George
How are you?
```

1.3.5.2 Variable scope: Local and global Variables

Local variable: When we declare variables inside a function, these variables will have a local scope (within the function). We cannot access them outside the function. These types of variables are called local variables. For example,

```python
def greet():
    message = 'Hello'  # local variable
    print('Local', message)
greet()
```

```
Local Hello
```

```python
print(message) # try to access message variable outside greet() function
```

```
NameError: name 'message' is not defined
```

As `message` was defined within the function `greet()`, it is local to the function, and cannot be called outside the function.

Global variable: Aa variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

Let's see an example of how a global variable is created.

```python
message = 'Hello'  # declare global variable

def greet():
    print('Local', message)  # declare local variable

greet()
print('Global', message)
```

```
Local Hello
Global Hello
```

1.3.5.3 Named arguments

Invoking a function with many arguments can often get confusing and is prone to human errors. Python provides the option of invoking functions with *named* arguments for better clarity. You can also split function invocation into multiple lines.

```
def loan_emi(amount, duration, rate, down_payment=0):
    loan_amount = amount - down_payment
    emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
    return emi
```

```
emi1 = loan_emi(
    amount=1260000,
    duration=8*12,
    rate=0.1/12,
    down_payment=3e5
)
```

```
emi1
```

```
14567.19753389219
```

1.3.5.4 Optional Arguments

Functions with optional arguments offer more flexibility in how you can use them. You can call the function with or without the argument, and if there is no argument in the function call, then a default value is used.

```
emi2 = loan_emi(
    amount=1260000,
    duration=8*12,
    rate=0.1/12)

emi2
```

```
19119.4467632335
```

1.3.5.5 *args and **kwargs

We can pass a variable number of arguments to a function using special symbols. There are two special symbols:

Special Symbols Used for passing arguments: 1. *args (Non-Keyword Arguments) 2. **kwargs (Keyword Arguments) > Note: "We use the "wildcard" or "" notation like this – *args OR **kwargs – as our function's argument when we have doubts about the number of arguments we should pass in a function."*

```python
def myFun(*args,**kwargs):
    print("args: ", args)
    print("kwargs: ", kwargs)


# Now we can use both *args ,**kwargs
# to pass arguments to this function :
myFun('John',22,'cs',name="John",age=22,major="cs")
```

```
args:  ('John', 22, 'cs')
kwargs:  {'name': 'John', 'age': 22, 'major': 'cs'}
```

1.3.6 Branching and looping (control flow)

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

1.3.6.1 Branching with `if`, `else` and `elif`

One of the most powerful features of programming languages is *branching*: the ability to make decisions and execute a different set of statements based on whether one or more conditions are true.

The `if` statement

In Python, branching is implemented using the `if` statement, which is written as follows:

```
if condition:
    statement1
    statement2
```

The `condition` can be a value, variable or expression. If the condition evaluates to `True`, then the statements within the *`if` block* are executed. Notice the four spaces before `statement1`, `statement2`, etc. The spaces inform Python that these statements are associated with the `if` statement above. This technique of structuring code by adding spaces is called *indentation*.

> **Indentation**: Python relies heavily on *indentation* (white space before a statement) to define code structure. This makes Python code easy to read and understand. You can run into problems if you don't use indentation properly. Indent your code by placing the cursor at the start of the line and pressing the `Tab` key once to add 4 spaces. Pressing `Tab` again will indent the code further by 4 more spaces, and press `Shift+Tab` will reduce the indentation by 4 spaces.

For example, let's write some code to check and print a message if a given number is even.

```
a_number = 34
```

```
if a_number % 2 == 0:
    print("We're inside an if block")
    print('The given number {} is even.'.format(a_number))
```

```
We're inside an if block
The given number 34 is even.
```

The `else` statement

We may want to print a different message if the number is not even in the above example. This can be done by adding the `else` statement. It is written as follows:

```
if condition:
    statement1
    statement2
else:
    statement4
    statement5
```

If `condition` evaluates to `True`, the statements in the `if` block are executed. If it evaluates to `False`, the statements in the `else` block are executed.

```
if a_number % 2 == 0:
    print('The given number {} is even.'.format(a_number))
else:
```

```python
    print('The given number {} is odd.'.format(a_number))
```

```
The given number 34 is even.
```

The `elif` statement

Python also provides an `elif` statement (short for "else if") to chain a series of conditional blocks. The conditions are evaluated one by one. For the first condition that evaluates to `True`, the block of statements below it is executed. The remaining conditions and statements are not evaluated. So, in an `if`, `elif`, `elif`… chain, at most one block of statements is executed, the one corresponding to the first condition that evaluates to `True`.

```python
today = 'Wednesday'
```

```python
if today == 'Sunday':
    print("Today is the day of the sun.")
elif today == 'Monday':
    print("Today is the day of the moon.")
elif today == 'Tuesday':
    print("Today is the day of Tyr, the god of war.")
elif today == 'Wednesday':
    print("Today is the day of Odin, the supreme diety.")
elif today == 'Thursday':
    print("Today is the day of Thor, the god of thunder.")
elif today == 'Friday':
    print("Today is the day of Frigga, the goddess of beauty.")
elif today == 'Saturday':
    print("Today is the day of Saturn, the god of fun and feasting.")
```

```
Today is the day of Odin, the supreme diety.
```

In the above example, the first 3 conditions evaluate to `False`, so none of the first 3 messages are printed. The fourth condition evaluates to `True`, so the corresponding message is printed. The remaining conditions are skipped. Try changing the value of `today` above and re-executing the cells to print all the different messages.

Using `if`, `elif`, and `else` together

You can also include an `else` statement at the end of a chain of `if`, `elif`… statements. This code within the `else` block is evaluated when none of the conditions hold true.

```python
a_number = 49
```

```python
if a_number % 2 == 0:
    print('{} is divisible by 2'.format(a_number))
elif a_number % 3 == 0:
    print('{} is divisible by 3'.format(a_number))
elif a_number % 5 == 0:
    print('{} is divisible by 5'.format(a_number))
else:
    print('All checks failed!')
    print('{} is not divisible by 2, 3 or 5'.format(a_number))
```

```
All checks failed!
49 is not divisible by 2, 3 or 5
```

Non-Boolean Conditions

Note that conditions do not necessarily have to be booleans. In fact, a condition can be any value. The value is converted into a boolean automatically using the `bool` operator. Any value in Python can be converted to a Boolean using the `bool` function.

Only the following values evaluate to `False` (they are often called *falsy* values):

1. The value `False` itself
2. The integer `0`
3. The float `0.0`
4. The empty value `None`
5. The empty text `""`
6. The empty list `[]`
7. The empty tuple `()`
8. The empty dictionary `{}`
9. The empty set `set()`
10. The empty range `range(0)`

Everything else evaluates to `True` (a value that evaluates to `True` is often called a *truthy* value).

```python
if '':
    print('The condition evaluted to True')
else:
    print('The condition evaluted to False')
```

```
The condition evaluated to False
```

```python
if 'Hello':
    print('The condition evaluated to True')
else:
    print('The condition evaluated to False')
```

```
The condition evaluated to True
```

```python
if { 'a': 34 }:
    print('The condition evaluated to True')
else:
    print('The condition evaluated to False')
```

```
The condition evaluated to True
```

```python
if None:
    print('The condition evaluated to True')
else:
    print('The condition evaluated to False')
```

```
The condition evaluated to False
```

Nested conditional statements

The code inside an `if` block can also include an `if` statement inside it. This pattern is called `nesting` and is used to check for another condition after a particular condition holds true.

```python
a_number = 15
```

```python
if a_number % 2 == 0:
    print("{} is even".format(a_number))
    if a_number % 3 == 0:
        print("{} is also divisible by 3".format(a_number))
    else:
        print("{} is not divisibule by 3".format(a_number))
else:
    print("{} is odd".format(a_number))
```

```python
    if a_number % 5 == 0:
        print("{} is also divisible by 5".format(a_number))
    else:
        print("{} is not divisibule by 5".format(a_number))
```

```
15 is odd
15 is also divisible by 5
```

Notice how the `print` statements are indented by 8 spaces to indicate that they are part of the inner `if`/`else` blocks.

> Nested `if`, `else` statements are often confusing to read and prone to human error. It's good to avoid nesting whenever possible, or limit the nesting to 1 or 2 levels.

Shorthand `if` conditional expression

A frequent use case of the `if` statement involves testing a condition and setting a variable's value based on the condition.

Python provides a shorter syntax, which allows writing such conditions in a single line of code. It is known as a *conditional expression*, sometimes also referred to as a *ternary operator*. It has the following syntax:

```
x = true_value if condition else false_value
```

It has the same behavior as the following `if-else` block:

```
if condition:
    x = true_value
else:
    x = false_value
```

Let's try it out for the example above.

```python
parity = 'even' if a_number % 2 == 0 else 'odd'
```

```python
print('The number {} is {}.'.format(a_number, parity))
```

```
The number 15 is odd.
```

The pass statement

if statements cannot be empty, there must be at least one statement in every if and elif block. We can use the pass statement to do nothing and avoid getting an error.

```
a_number = 9
```

```
if a_number % 2 == 0:

elif a_number % 3 == 0:
    print('{} is divisible by 3 but not divisible by 2')
```

```
IndentationError: expected an indented block (1562158884.py, line 3)
```

As there must be at least one statement withihng the if block, the above code throws an error.

```
if a_number % 2 == 0:
    pass
elif a_number % 3 == 0:
    print('{} is divisible by 3 but not divisible by 2'.format(a_number))
```

```
9 is divisible by 3 but not divisible by 2
```

1.3.6.2 Iteration with while loops

Another powerful feature of programming languages, closely related to branching, is running one or more statements multiple times. This feature is often referred to as *iteration* on *looping*, and there are two ways to do this in Python: using while loops and for loops.

while loops have the following syntax:

```
while condition:
    statement(s)
```

Statements in the code block under while are executed repeatedly as long as the condition evaluates to True. Generally, one of the statements under while makes some change to a variable that causes the condition to evaluate to False after a certain number of iterations.

Let's try to calculate the factorial of 100 using a while loop. The factorial of a number n is the product (multiplication) of all the numbers from 1 to n, i.e., 1*2*3*...*(n-2)*(n-1)*n.

```python
result = 1
i = 1

while i <= 10:
    result = result * i
    i = i+1

print('The factorial of 100 is: {}'.format(result))
```

```
The factorial of 100 is: 3628800
```

1.3.6.3 Infinite Loops

Suppose the condition in a `while` loop always holds true. In that case, Python repeatedly executes the code within the loop forever, and the execution of the code never completes. This situation is called an infinite loop. It generally indicates that you've made a mistake in your code. For example, you may have provided the wrong condition or forgotten to update a variable within the loop, eventually falsifying the condition.

If your code is *stuck* in an infinite loop during execution, just press the "Stop" button on the toolbar (next to "Run") or select "Kernel > Interrupt" from the menu bar. This will *interrupt* the execution of the code. The following two cells both lead to infinite loops and need to be interrupted.

```python
# INFINITE LOOP - INTERRUPT THIS CELL

result = 1
i = 1

while i <= 100:
    result = result * i
    # forgot to increment i
```

```python
# INFINITE LOOP - INTERRUPT THIS CELL

result = 1
i = 1

while i > 0 : # wrong condition
    result *= i
    i += 1
```

1.3.6.4 `break` and `continue` statements

In Python, `break` and `continue` statements can alter the flow of a normal loop.

We can use the `break` statement within the loop's body to immediately stop the execution and *break* out of the loop. with the continue statement. If the condition evaluates to `True`, then the loop will move to the next iteration.

```
i = 1
result = 1

while i <= 100:
    result *= i
    if i == 42:
        print('Magic number 42 reached! Stopping execution..')
        break
    i += 1

print('i:', i)
print('result:', result)
```

```
Magic number 42 reached! Stopping execution..
i: 42
result: 1405006117752879898543142606244511569936384000000000
```

```
i = 1
result = 1

while i < 8:
    i += 1
    if i % 2 == 0:
        print('Skipping {}'.format(i))
        continue
    print('Multiplying with {}'.format(i))
    result = result * i

print('i:', i)
print('result:', result)
```

```
Skipping 2
Multiplying with 3
```

```
Skipping 4
Multiplying with 5
Skipping 6
Multiplying with 7
Skipping 8
i: 8
result: 105
```

In the example above, the statement `result = result * i` inside the loop is skipped when `i` is even, as indicated by the messages printed during execution.

> **Logging**: The process of adding `print` statements at different points in the code *(often within loops and conditional statements)* for inspecting the values of variables at various stages of execution is called logging. As our programs get larger, they naturally become prone to human errors. Logging can help in verifying the program is working as expected. In many cases, `print` statements are added while writing & testing some code and are removed later.

Task: Guess the output and explain it.

```python
# Use of break statement inside the loop

for val in "string":
    if val == "i":
        break
    print(val)

print("The end")
```

```
s
t
r
The end
```

```python
# Program to show the use of continue statement inside loops

for val in "string":
    if val == "i":
        continue
    print(val)

print("The end")
```

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

1.3.6.5 Iteration with `for` loops

A `for` loop is used for iterating or looping over sequences, i.e., lists, tuples, dictionaries, strings, and *ranges*. For loops have the following syntax:

```
for value in sequence:
    statement(s)
```

The statements within the loop are executed once for each element in `sequence`. Here's an example that prints all the element of a list.

```python
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

for day in days:
    print(day)
```

```
Monday
Tuesday
Wednesday
Thursday
Friday
```

```python
# Looping over a string
for char in 'Monday':
    print(char)
```

```
M
o
n
d
a
y
```

```python
# Looping over a dictionary
person = {
    'name': 'John Doe',
    'sex': 'Male',
    'age': 32,
    'married': True
}

for key, value in person.items():
    print("Key:", key, ",", "Value:", value)
```

```
Key: name , Value: John Doe
Key: sex , Value: Male
Key: age , Value: 32
Key: married , Value: True
```

1.3.7 Iterating using `range` and `enumerate`

The `range` function is used to create a sequence of numbers that can be iterated over using a `for` loop. It can be used in 3 ways:

- `range(n)` - Creates a sequence of numbers from `0` to `n-1`
- `range(a, b)` - Creates a sequence of numbers from `a` to `b-1`
- `range(a, b, step)` - Creates a sequence of numbers from `a` to `b-1` with increments of `step`

Let's try it out.

```python
for i in range(4):
    print(i)
```

```
0
1
2
3
```

```python
for i in range(3, 8):
    print(i)
```

```
3
4
5
6
7
```

```
for i in range(3, 14, 4):
    print(i)
```

```
3
7
11
```

Ranges are used for iterating over lists when you need to track the index of elements while iterating.

```
a_list = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

for i in range(len(a_list)):
    print('The value at position {} is {}.'.format(i, a_list[i]))
```

```
The value at position 0 is Monday.
The value at position 1 is Tuesday.
The value at position 2 is Wednesday.
The value at position 3 is Thursday.
The value at position 4 is Friday.
```

Another way to achieve the same result is by using the `enumerate` function with `a_list` as an input, which returns a tuple containing the index and the corresponding element.

```
for i, val in enumerate(a_list):
    print('The value at position {} is {}.'.format(i, val))
```

```
The value at position 0 is Monday.
The value at position 1 is Tuesday.
The value at position 2 is Wednesday.
The value at position 3 is Thursday.
The value at position 4 is Friday.
```

2 Data structures

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

2.1 Tuple

Tuple is a sequence of python objects, with two key characteristics: (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 in-built python function *type()*. 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.

2.1.1 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')
```

2.1.2 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:

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

```
x
```

```
4.5
```

```
y
```

```
'99'
```

```
z
```

```
99
```

2.1.3 Tuple methods

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

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

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

```
3
```

```
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>
```

2.2 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]
```

2.2.1 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]
```

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

2.2.2 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
```

2.2.3 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:

2.2.3.1

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
```

2.2.3.2

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
```

2.2.3.3

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 %
```

2.2.3.4

Redo Q2.2.3.3 using the if-else statement within list comprehension.

2.2.4 Practice exercise 2

Below is a list consisting of responses to the question: "What do you expect your starting salary to be after graduation, to the nearest thousand dollars? (ex: 47000)" from students of the STAT303-1 Fall 2023. class.

```
expected_salary = ['90000', '110000', '100000', '90k', '80000', '47000', '100000', '70000'
```

Clean `expected_salary` using list comprehensions only, and find the mean expected salary.

2.2.5 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]
```

2.2.6 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]
```

2.2.7 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]
```

2.2.8 Practice exercise 3

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

2.2.8.1

Set the second entry (index 1) to 17

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

2.2.8.2

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

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

2.2.8.3

Remove the first entry from the list

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

```
8
```

2.2.8.4

Sort the list

```
L.sort()
```

2.2.8.5

Double the list (concatenate the list to itself)

```
L=L+L
```

2.2.8.6

Insert 25 at index 3

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

```
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 advantages 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 original 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
```

2.3 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'
```

2.3.1 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'}
```

2.3.2 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
```

2.3.3 Practice exercise 4

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
```

2.4 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.

2.4.1 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
```

2.4.2 Practice exercise 5

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):

    #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 %
```

2.5 Practice exercise 6

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.

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

Use the object above to answer the following questions:

2.5.1

What is the datatype of the object?

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

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

2.5.1.1

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

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

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

2.5.1.2

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'>
```

2.5.1.3

How many calories are there in `Iced Coffee`?

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

```
Calories =  5
```

2.5.1.4

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}
```

2.5.1.5

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}
```

2.5.1.6

Answer Q2.5.1.5 using only dictionary comprehension.

Part II

Exploratory data analysis

3 Reading data

3.1 Types of data - structured and unstructured

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.

3.2 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.

3.2.1 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.

3.2.2 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')
```

3.2.3 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

3.2.3.1 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**.

3.2.3.2 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).

3.2.4 Summary statistics

3.2.4.1 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 Year
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

	US Gross	Worldwide Gross	Production Budget	IMDB Rating	IMDB Votes	Release Yea
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>
```

3.2.4.2 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
```

3.2.5 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.

3.2.5.1

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

3.2.5.2

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

3.2.5.3

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
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
```

3.2.6 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

3.2.7 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
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

3.2.8 Practice exercise 2

3.2.8.1

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`

3.2.8.2

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

3.2.8.3

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

3.2.9 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()
```

Rank	Title	US Gross	Worldwide Gross	Production Budget	Release Date	MI
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.

3.2.10 Practice exercise 3

3.2.10.1

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.

3.2.10.2

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
```

3.2.10.3

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.

3.3 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*.

3.3.1 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.

3.3.2 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...

;Unnamed: 0;Name;Author;User Rating;Reviews;Price;Year;Genre

```
#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
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

3.3.3 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

	Country/Territory	UN Region	IMF[4][5]		United Nations[6]		World Bank[7]	
	Country/Territory	UN Region	Estimate	Year	Estimate	Year	Estimate	Year
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 keywords 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.

3.3.4 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
```

3.3.5 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

3.3.6 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
```

3.3.7 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.

3.4 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
```

```
#Using the NumPy function array() to define a NumPy array
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 structures 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:

4.2.1 `ndim`

Shows the number of dimensions (or axes) of the array.

```
numpy_ex.ndim
```

```
2
```

4.2.2 `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)
```

4.2.3 `size`

This is the total number of elements of the array, which is the product of the elements of shape.

```
numpy_ex.size
```

```
6
```

4.2.4 `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')
```

4.2.5 T

This attribute is used to transpose the NumPy array. This is often used to make matrices (2-dimensional arrays) compatible for multiplication.

For matrix multiplication, the columns of the first matrix must be equal to the rows of the second matrix. For example, consider the matrix below:

```
matrix_to_multiply = np.array([[1,2,1],[0,1,0]])
```

Suppose we wish to multiply this matrix with `numpy_ex`. Note the shape of both the matrices below.

```
matrix_to_multiply.shape
```

```
(2, 3)
```

```
numpy_ex.shape
```

```
(2, 3)
```

To multiply the above matrices the number of columns of the one of the matrices must be the same as the number of rows of the other matrix. With the current matrices, this is not true as the number of columns of the first matrix is 3, and the the number of rows of the second matrix is 2 (no matter which matrix is considered to be the first one).

However, if we transpose one of the matrices, their shapes will be compatible for multiplication. Let's transpose `matrix_to_multiply`:

```
matrix_to_multiply_transpose = matrix_to_multiply.T
matrix_to_multiply_transpose
```

```
array([[1, 0],
       [2, 1],
       [1, 0]])
```

The shape of `matrix_to_multiply_transpose` is:

```
matrix_to_multiply_transpose.shape
```

```
(3, 2)
```

The matrices `matrix_to_multiply_transpose` and `numpy_ex` are compatible for matrix multiplication. However, the result will depend on the order in which the matrices are multiplied:

```
#Matrix multiplication with matrix_to_multiply_transpose before numpy_ex
matrix_to_multiply_transpose.dot(numpy_ex)
```

```
array([[ 1,  2,  3],
       [ 6,  9, 12],
       [ 1,  2,  3]])
```

```
#Matrix multiplication with numpy_ex before matrix_to_multiply_transpose
numpy_ex.dot(matrix_to_multiply_transpose)
```

```
array([[ 8,  2],
       [20,  5]])
```

The shape of the resulting matrix is equal to the rows of the first matrix and the columns of the second matrix. The order of matrices must be decided as per the requirements of the problem.

4.3 Arithmetic 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 arithmetic operation on an array with an array of a different shape.

Below are some examples of arithmetic operations on arrays.

```
#Defining two arrays of the same shape
arr1 = np.array([[1, 2, 3, 4],
                 [5, 6, 7, 8],
```

```python
                 [9, 1, 2, 3]])
arr2 = np.array([[11, 12, 13, 14],
                 [15, 16, 17, 18],
                 [19, 11, 12, 13]])
```

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

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

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

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

```python
# 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]])
```

```
# 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.

```
arr1 = np.array([[1, 2, 3, 4],
                 [5, 6, 7, 8],
                 [9, 1, 2, 3]])
arr2 = np.array([4, 5, 6, 7])
```

```
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`:

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

```
#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.

```python
#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')
```

```python
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

```python
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

```python
#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.

Solution:

```
import pandas as pd
data = pd.read_csv('./Datasets/movies_cleaned.csv')
data.head()
```

	Title	IMDB Rating	Rotten Tomatoes Rating	Running Time min	Release Date	US (
0	Broken Arrow	5.8	55	108	Feb 09 1996	7064
1	Brazil	8.0	98	136	Dec 18 1985	9929
2	The Cable Guy	5.8	52	95	Jun 14 1996	6024
3	Chain Reaction	5.2	13	106	Aug 02 1996	2122
4	Clash of the Titans	5.9	65	108	Jun 12 1981	3000

```
# Getting ratings of all movies
drating = data[['IMDB Rating','Rotten Tomatoes Rating']]
drating_num = drating.to_numpy() #Converting the data to NumPy array
drating_num
```

```
array([[ 5.8, 55. ],
       [ 8. , 98. ],
       [ 5.8, 52. ],
       ...,
       [ 7. , 65. ],
       [ 5.7, 26. ],
       [ 6.7, 82. ]])
```

```
# Getting the matrix indicating the genre of all movies
dgenre = data.iloc[:,8:12]
dgenre_num = dgenre.to_numpy() #Converting the data to NumPy array
dgenre_num
```

```
array([[0, 1, 0, 0],
       [1, 0, 0, 0],
       [1, 0, 0, 0],
       ...,
       [1, 0, 0, 0],
```

```
       [0, 1, 0, 0],
       [0, 1, 0, 0]], dtype=int64)
```

We'll first find the total IMDB and Rotten tomatoes ratings for all movies of each genre, and then divide them by the number of movies of the corresponding genre to find the average rating for the genre.

For finding the total IMDB and Rotten tomatoes ratings, we'll multiply `drating_num` with `dgenre_num`. However, before multiplying, we'll check if their shapes are compatible for matrix multiplication.

```
#Shape of drating_num
drating_num.shape
```

```
(980, 2)
```

```
#Shape of dgenre_num
dgenre_num.shape
```

```
(980, 4)
```

Note that the above shapes are not compatible for matrix multiplication. We'll transpose `dgenre_num` to make the shapes compatible.

```
#Total IMDB and Rotten tomatoes ratings for each genre
ratings_sum_genre = drating_num.T.dot(dgenre_num)
ratings_sum_genre
```

```
array([[ 1785.6,  1673.1,  1630.3,   946.2],
       [14119. , 13725. , 14535. ,  6533. ]])
```

```
#Number of movies in the data will be stored in 'rows', and number of columns stored in 'c
rows, cols = data.shape
```

```
#Getting number of movies in each genre
movies_count_genre = dgenre_num.T.dot(np.ones(rows))
movies_count_genre
```

```
array([302., 264., 239., 154.])
```

```
#Finding the average IMDB and average Rotten tomatoes ratings for each genre
ratings_sum_genre/movies_count_genre
```

```
array([[ 5.91258278,  6.3375    ,  6.82133891,  6.14415584],
       [46.75165563, 51.98863636, 60.81589958, 42.42207792]])
```

```
pd.DataFrame(ratings_sum_genre/movies_count_genre,columns = ['comedy','Action','drama','ho
             index = ['IMDB Rating','Rotten Tomatoes Rating'])
```

	comedy	Action	drama	horror
IMDB Rating	5.912583	6.337500	6.821339	6.144156
Rotten Tomatoes Rating	46.751656	51.988636	60.815900	42.422078

IMDB users prefer *drama*, and are amused the least by *comedy* movies, on an average. However, Rotten tomatoes critics would rather watch *comedy* than *horror* movies, on an average.

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 =  16.22666
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.

Solution:

```
#Reading data
movies = pd.read_csv('./Datasets/movies_cleaned.csv')
```

```
#Filtering action movies
movies_action = movies.loc[movies['Action']==1,:]

#Computing profit of movies
movies_action.loc[:,'Profit'] = movies_action.loc[:,'Worldwide Gross'] - movies_action.loc

#Subsetting the profit column
profit_vec = movies_action['Profit']

#Creating a matrix of 1000 samples with replacement from the profit column
bootstrap_samples=np.random.choice(profit_vec,size = (1000,len(profit_vec)))

#Computing the mean of each of the 1000 samples
bootstrap_sample_means = bootstrap_samples.mean(axis=1)

#The confidence interval is the 2.5th and 97.5th percentile of the mean of the 1000 sample
print("Confidence interval = [$"+str(np.round(np.percentile(bootstrap_sample_means,2.5)/1e
```

```
Confidence interval = [$132.53 million, $182.69 million]
```

5 Pandas

5.1 Introduction

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 distribution
- Visualizing the data with help from the Matplotlib library
- Writing the cleaned and transformed data into a CSV file or other database formats

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

```
import pandas as pd
```

5.2 Pandas data structures - Series and DataFrame

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

A DataFrame is a two-dimensional object - comprising 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 labels (also called row indices) which refer to individual rows, and column labels (also called column names) that refer to individual 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 = spotify_data['track_name']
spotify_songs
```

```
0                           All Girls Are The Same
1                                     Lucid Dreams
2                                  Hear Me Calling
3                                          Robbery
4                                      Big Stepper
                            ...
243185                                    Stardust
```

```
243186              Knockin' A Jug - 78 rpm Version
243187            When It's Sleepy Time Down South
243188    On The Sunny Side Of The Street - Part 2
243189                                    My Sweet
Name: track_name, Length: 243190, 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>
```

5.3 Creating a Pandas Series / DataFrame

5.3.1 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
```

5.3.2 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

5.3.3 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.

5.4 Attributes and Methods of a Pandas DataFrame

All attributes and methods of a Pandas DataFrame object can be viewed with the python's built-in dir() function.

```
#List of attributes and methods of a Pandas DataFrame
#This code is not executed as the list is too long
dir(spotify_data)
```

Although we'll see examples of attributes and methods of a Pandas DataFrame, please note that most of these attributes and methods are also applicable to the Pandas Series object.

5.4.1 Attributes of a Pandas DataFrame

Some of the attributes of the Pandas DataFrame class are the following.

5.4.1.1 `dtypes`

This attribute is a Series consisting the datatypes of columns of a Pandas DataFrame.

```
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
```

The table below 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 having a maximum size 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

5.4.1.2 `columns`

This attribute consists of the column labels (or column names) of a Pandas DataFrame.

```
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')
```

5.4.1.3 `index`

This attribute consists of the row lables (or row indices) of a Pandas DataFrame.

```
spotify_data.index
```

```
RangeIndex(start=0, stop=243190, step=1)
```

5.4.1.4 axes

This is a list of length two, where the first element is the row labels, and the second element is the columns labels. In other words, this attribute combines the information in the attributes - index and columns.

```
spotify_data.axes
```

```
[RangeIndex(start=0, stop=243190, step=1),
 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')]
```

5.4.1.5 ndim

As in NumPy, this attribute specifies the number of dimensions. However, unlike NumPy, a Pandas DataFrame has a fixed dimenstion of 2, and a Pandas Series has a fixed dimesion of 1.

```
spotify_data.ndim
```

```
2
```

5.4.1.6 size

This attribute specifies the number of elements in a DataFrame. Its value is the product of the number of rows and columns.

```
spotify_data.size
```

```
5106990
```

5.4.1.7 `shape`

This is a tuple consisting of the number of rows and columns in a Pandas DataFrame.

```
spotify_data.shape
```

```
(243190, 21)
```

5.4.1.8 `values`

This provides a NumPy representation of a Pandas DataFrame.

```
spotify_data.values
```

```
array([[16996777, 'rap', 'Juice WRLD', ..., 0.203, 161.991, 4],
       [16996777, 'rap', 'Juice WRLD', ..., 0.218, 83.903, 4],
       [16996777, 'rap', 'Juice WRLD', ..., 0.499, 88.933, 4],
       ...,
       [2256652, 'jazz', 'Louis Armstrong', ..., 0.37, 105.093, 4],
       [2256652, 'jazz', 'Louis Armstrong', ..., 0.576, 101.279, 4],
       [2256652, 'jazz', 'Louis Armstrong', ..., 0.816, 105.84, 4]],
      dtype=object)
```

5.4.2 Methods of a Pandas DataFrame

Some of the commonly used methods of the Pandas DataFrame class are the following.

5.4.2.1 `head()`

Prints the first n rows of a DataFrame.

```
spotify_data.head(2)
```

	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

5.4.2.2 tail()

Prints the last n rows of a DataFrame.

```
spotify_data.tail(3)
```

	artist_followers	genres	artist_name	artist_popularity	track_name
243187	2256652	jazz	Louis Armstrong	74	When It's Sleepy Time Down Sout
243188	2256652	jazz	Louis Armstrong	74	On The Sunny Side Of The Street
243189	2256652	jazz	Louis Armstrong	74	My Sweet

5.4.2.3 describe()

Print summary statistics of a Pandas DataFrame, as seen in chapter 3 on Reading Data.

```
spotify_data.describe()
```

	artist_followers	artist_popularity	track_popularity	duration_ms	explicit	release_yea
count	2.431900e+05	243190.000000	243190.000000	2.431900e+05	243190.000000	243190.0000
mean	1.960931e+06	65.342633	36.080772	2.263209e+05	0.050039	1992.475258
std	5.028746e+06	10.289182	16.476836	9.973214e+04	0.218026	18.481463
min	2.300000e+01	51.000000	0.000000	3.344000e+03	0.000000	1923.000000
25%	1.832620e+05	57.000000	25.000000	1.776670e+05	0.000000	1980.000000
50%	5.352520e+05	64.000000	36.000000	2.188670e+05	0.000000	1994.000000
75%	1.587332e+06	72.000000	48.000000	2.645465e+05	0.000000	2008.000000
max	7.890023e+07	100.000000	99.000000	4.995083e+06	1.000000	2021.000000

5.4.2.4 max()/min()

Returns the max/min values of numeric columns. If the function is applied on non-numeric columns, it will return the maximum/minimum value based on the order of the alphabet.

```
#The max() method applied on a Series
spotify_data['artist_popularity'].max()
```

```
100
```

```
#The max() method applied on a DataFrame
spotify_data.max()
```

```
artist_followers                    78900234
genres                                  rock
artist_name                             OSN 
artist_popularity                        100
track_name                      days gone by
track_popularity                          99
duration_ms                          4995083
explicit                                   1
release_year                            2021
danceability                           0.988
energy                                   1.0
key                                       11
loudness                               3.744
mode                                       1
speechiness                            0.969
acousticness                           0.996
instrumentalness                         1.0
liveness                                 1.0
valence                                  1.0
tempo                                243.507
time_signature                             5
dtype: object
```

5.4.2.5 `mean()`/`median()`

Returns the mean/median values of numeric columns.

```
spotify_data.median()
```

```
artist_followers     535252.000000
artist_popularity        64.000000
track_popularity         36.000000
duration_ms          218867.000000
explicit                  0.000000
release_year           1994.000000
danceability              0.579000
energy                    0.591000
```

```
key                        5.000000
loudness                  -8.645000
mode                       1.000000
speechiness                0.043100
acousticness               0.325000
instrumentalness           0.000011
liveness                   0.141000
valence                    0.560000
tempo                    118.002000
time_signature             4.000000
dtype: float64
```

5.4.2.6 `std()`

Returns the standard deviation of numeric columns.

```
spotify_data.std()
```

```
artist_followers     5.028746e+06
artist_popularity    1.028918e+01
track_popularity     1.647684e+01
duration_ms          9.973214e+04
explicit             2.180260e-01
release_year         1.848146e+01
danceability         1.594436e-01
energy               2.366309e-01
key                  3.532546e+00
loudness             4.449731e+00
mode                 4.698771e-01
speechiness          1.980684e-01
acousticness         3.211417e-01
instrumentalness     2.095551e-01
liveness             1.980759e-01
valence              2.500172e-01
tempo                2.986422e+01
time_signature       4.580822e-01
dtype: float64
```

5.4.2.7 `sample(n)`

Returns n random observations from a Pandas DataFrame.

```
spotify_data.sample(4)
```

	artist_followers	genres	artist_name	artist_popularity	track_name
42809	385756	rock	Saxon	56	Never Surrender - 2009 Remastered
25730	810526	hip hop	Froid	68	Pseudosocial
147392	479209	jazz	Sarah Vaughan	59	Love Dance
233189	1201905	rock	Grateful Dead	72	Cold Rain and Snow - 2013 Remast

5.4.2.8 dropna()

Drops all observations with at least one missing value.

```
#This code is not executed to avoid prining a large table
spotify_data.dropna()
```

5.4.2.9 apply()

This method is used to apply a function over all columns or rows of a Pandas DataFrame. For example, let us find the range of values of `artist_followers`, `artist_popularity` and `release_year`.

```
#Defining the function to compute range of values of a columns
def range_of_values(x):
    return x.max()-x.min()

#Applying the function to three coluumns for which we wish to find the range of values
spotify_data[['artist_followers','artist_popularity','release_year']].apply(range_of_value
```

```
artist_followers     78900211
artist_popularity          49
release_year               98
dtype: int64
```

The `apply()` method is often used with the one line function known as `lambda` function in python. These functions do not require a name, and can be defined using the keyword `lambda`. The above block of code can be concisely written as:

```
spotify_data[['artist_followers','artist_popularity','release_year']].apply(lambda x:x.max
```

```
artist_followers     78900211
artist_popularity          49
release_year               98
dtype: int64
```

Note that the Series object also has an apply() method associated with it. The method can be used to apply a function to each value of a Series.

5.4.2.10 `map()`

The function is used to map distinct values of a Pandas Series to another set of corresponding values.

For example, suppose we wish to create a new column in the spotify dataset which indicates the modality of the song - major (mode = 1) or minor (mode = 0). We'll map the values of the `mode` column to the categories *major* and *minor*:

```
#Creating a dictionary that maps the values 0 and 1 to minor and major respectively
map_mode = {0:'minor', 1:'major'}

#The map() function requires a dictionary object, and maps the 'values' of the 'keys' in t
spotify_data['modality'] = spotify_data['mode'].map(map_mode)
```

We can see the variable `modality` in the updated DataFrame.

```
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

5.4.2.11 `drop()`

This function is used to drop rows/columns from a DataFrame.

For example, let us drop the columns `mode` from the spotify dataset:

```
#Dropping the column 'mode'
spotify_data_new = spotify_data.drop('mode',axis=1)
spotify_data_new.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

Note that if multiple columns or rows are to be dropped, they must be enclosed in box brackets.

5.4.2.12 `unique()`

This functions 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)
```

5.4.2.13 `value_counts()`

This function 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*.

5.4.2.14 `isin()`

This function 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 Data manipulations with Pandas

5.5.1 Sub-setting data

5.5.1.1 `loc` and `iloc` with the original row / column index

Subsetting observations: 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, i.e., the index before the `,`. 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

Subsetting columns: 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

Note that when multiple columns are subset with `loc` they are enclosed in a box bracket, unlike the case with a single column. Similarly if multiple observations are selected using the row labels, the row labels must be enclosed in box brackets.

5.5.1.2 Re-indexing rows followed by `loc` / `iloc`

Suppose we wish to subset data based on the `genres`. If we want to subset *hiphop* songs, we may subset as:

```
#Subsetting hiphop songs
hiphop_songs = spotify_data.loc[spotify_data['genres']=='hip hop',:]
hiphop_songs.head()
```

	artist_followers	genres	artist_name	artist_popularity	track_name	track_popularity
64	6485079	hip hop	DaBaby	93	FIND MY WAY	0
80	22831280	hip hop	Daddy Yankee	91	Hula Hoop	0
81	22831280	hip hop	Daddy Yankee	91	Gasolina - Live	0
87	22831280	hip hop	Daddy Yankee	91	La Nueva Y La Ex	0
88	22831280	hip hop	Daddy Yankee	91	Que Tire Pa Lante	0

However, if we need to subset data by `genres` frequently in our analysis, and we don't need the current row labels, we may replace the row labels as `genres` to shorten the code for filtering the observations based on `genres`.

We use the set_index() function to re-index the rows based on existing column(s) of the DataFrame.

```
#Defining row labels as the values of the column `genres`
spotify_data_reindexed = spotify_data.set_index(keys=spotify_data['genres'])
spotify_data_reindexed.head()
```

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

Now, we can subset *hiphop* songs using the row label of the data:

```
#Subsetting hiphop songs using row labels
hiphop_songs = spotify_data_reindexed.loc['hip hop',:]
hiphop_songs.head()
```

	artist_followers	genres	artist_name	artist_popularity	track_name	track_popular
genres						
hip hop	6485079	hip hop	DaBaby	93	FIND MY WAY	0
hip hop	22831280	hip hop	Daddy Yankee	91	Hula Hoop	0
hip hop	22831280	hip hop	Daddy Yankee	91	Gasolina - Live	0
hip hop	22831280	hip hop	Daddy Yankee	91	La Nueva Y La Ex	0
hip hop	22831280	hip hop	Daddy Yankee	91	Que Tire Pa Lante	0

5.5.2 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>
```

5.5.3 Ranking data

With the rank() function, we can rank the observations.

For example, let us add a new column to the spotify data that provides the rank of the `track_popularity` column:

```
spotify_ranked = spotify_data.copy()
spotify_ranked['track_popularity_rank']=spotify_sorted['track_popularity'].rank()
spotify_ranked.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

Note the column `track_popularity_rank`. Why does it contain floating point numbers? Check the `rank()` documentation to find out!

5.5.4 Practice exercise 1

5.5.4.1

Read the file *STAT303-1 survey for data analysis.csv.*

```
survey_data = pd.read_csv('./Datasets/STAT303-1 survey for data analysis.csv')
```

5.5.4.2

How many observations and variables are there in the data?

```
print("The data has ",survey_data.shape[0],"observations, and", survey_data.shape[1], "col
```

```
The data has  192 observations, and 51 columns
```

5.5.4.3

Rename all the columns of the data, except the first two columns, with the shorter names in the list `new_col_names` given below. The order of column names in the list is the same as the order in which the columns are to be renamed starting with the third column from the left.

```
new_col_names = ['parties_per_month', 'do_you_smoke', 'weed', 'are_you_an_introvert_or_ext
```

```
survey_data.columns = list(survey_data.columns[0:2])+new_col_names
```

5.5.4.4

Rename the following columns again:

1. Rename `do_you_smoke` to `smoke`.
2. Rename `are_you_an_introvert_or_extrovert` to `introvert_extrovert`.

Hint: Use the function rename()

```
survey_data.rename(columns={'do_you_smoke':'smoke','are_you_an_introvert_or_extrovert':'in
```

5.5.4.5

Find the proportion of people going to more than 4 parties per month. Use the variable `parties_per_month`.

```
survey_data['parties_per_month']=pd.to_numeric(survey_data.parties_per_month,errors='coerc
survey_data.loc[survey_data['parties_per_month']>4,:].shape[0]/survey_data.shape[0]
```

```
0.3385416666666667
```

5.5.4.6

Among the people who go to more than 4 parties a month, what proportion of them are introverts?

```
survey_data.loc[((survey_data['parties_per_month']>4) & (survey_data.introvert_extrovert==
```

```
0.5076923076923077
```

5.5.4.7

Find the proportion of people in each category of the variable `how_happy`.

```
survey_data.how_happy.value_counts()/survey_data.shape[0]
```

```
Pretty happy     0.703125
Very happy       0.151042
Not too happy    0.088542
Don't know       0.057292
Name: how_happy, dtype: float64
```

5.5.4.8

Among the people who go to more than 4 parties a month, what proportion of them are either `Pretty happy` or `Very happy`?

```
survey_data.loc[((survey_data['parties_per_month']>4) & (survey_data.how_happy.isin(['Pret
```

```
0.9076923076923077
```

5.5.4.9

Examine the column `num_insta_followers`. Some numbers in the column contain a comma(`,`) or a tilde(`~`). Remove both these characters from the numbers in the column.

Hint: You may use the function str.replace() of the Pandas Series class.

```
survey_data_insta = survey_data.copy()
survey_data_insta['num_insta_followers']=survey_data_insta['num_insta_followers'].str.repl
survey_data_insta['num_insta_followers']=survey_data_insta['num_insta_followers'].str.repl
```

5.5.4.10

Convert the column `num_insta_followers` to numeric. Coerce the errors.

```
survey_data_insta.num_insta_followers = pd.to_numeric(survey_data_insta.num_insta_follower
```

5.5.4.11

Drop the observations consisting of missing values for `num_insta_followers`. Report the number of observations dropped.

```
survey_data.num_insta_followers.isna().sum()
```

```
3
```

There are 3 missing values of `num_insta_followers`.

```
#Dropping observations with missing values of num_insta_followers
survey_data=survey_data[~survey_data.num_insta_followers.isna()]
```

5.5.4.12

What is the mean `internet_hours_per_day` for the top 46 people in terms of number of instagram followers?

```
survey_data_insta.sort_values(by = 'num_insta_followers',ascending=False, inplace=True)
top_insta = survey_data_insta.iloc[:46,:]
top_insta.internet_hours_per_day = pd.to_numeric(top_insta.internet_hours_per_day,errors =
top_insta.internet_hours_per_day.mean()
```

```
5.088888888888889
```

5.5.4.13

What is the mean `internet_hours_per_day` for the remaining people?

```
low_insta = survey_data_insta.iloc[46:,:]
low_insta.internet_hours_per_day = pd.to_numeric(low_insta.internet_hours_per_day,errors =
low_insta.internet_hours_per_day.mean()
```

```
13.118881118881118
```

5.6 Arithematic operations

5.6.1 Arithematic operations between DataFrames

Let us create two toy DataFrames:

```
#Creating two toy DataFrames
toy_df1 = pd.DataFrame([(1,2),(3,4),(5,6)], columns=['a','b'])
toy_df2 = pd.DataFrame([(100,200),(300,400),(500,600)], columns=['a','b'])
```

```
#DataFrame 1
toy_df1
```

	a	b
0	1	2
1	3	4
2	5	6

```
#DataFrame 2
toy_df2
```

	a	b
0	100	200
1	300	400
2	500	600

Element by element operations between two DataFrames can be performed with the operators `+`, `-`, `*`,`/`,`**`, and `%`. Below is an example of element-by-element addition of two DataFrames:

```
# Element-by-element arithmetic addition of the two DataFrames
toy_df1 + toy_df2
```

	a	b
0	101	202
1	303	404
2	505	606

Note that these operations create problems when the row indices and/or column names of the two DataFrames do not match. See the example below:

```
#Creating another toy example of a DataFrame
toy_df3 = pd.DataFrame([(100,200),(300,400),(500,600)], columns=['a','b'], index=[1,2,3])
toy_df3
```

	a	b
1	100	200
2	300	400
3	500	600

```
#Adding DataFrames with some unmatching row indices
toy_df1 + toy_df3
```

	a	b
0	NaN	NaN
1	103.0	204.0
2	305.0	406.0
3	NaN	NaN

Note that the rows whose indices match between the two DataFrames are added up. The rest of the values are missing (or `NaN`) because only one of the DataFrames has that index.

As in the case of row indices, missing values will also appear in the case of unmatching column names, as shown in the example below.

```
toy_df4 = pd.DataFrame([(100,200),(300,400),(500,600)], columns=['b','c'])
toy_df4
```

	b	c
0	100	200
1	300	400
2	500	600

```
#Adding DataFrames with some unmatching column names
toy_df1 + toy_df4
```

	a	b	c
0	NaN	102	NaN
1	NaN	304	NaN
2	NaN	506	NaN

5.6.2 Arithematic operations between a Series and a DataFrame

Broadcasting: As in NumPy, we can **broadcast** a Series to match the shape of another DataFrame:

```
# Broadcasting: The row [1,2] (a Series) is added on every row in df2
toy_df1.iloc[0,:] + toy_df2
```

	a	b
0	101	202
1	301	402
2	501	602

Note that the `+` operator is used to add values of a Series to a DataFrame based on column names. For adding a Series to a DataFrame based on row indices, we cannot use the `+` operator. Instead, we'll need to use the `add()` function as explained below.

Broadcasting based on row/column labels: We can use the add() function to broadcast a Series to a DataFrame. By default the Series adds based on column names, as in the case of the `+` operator.

```
# Add the first row of df1 (a Series) to every row in df2
toy_df2.add(toy_df1.iloc[0,:])
```

	a	b
0	101	202
1	301	402
2	501	602

For broadcasting based on row indices, we use the `axis` argument of the `add()` function.

```
# The second column of df1 (a Series) is added to every col in df2
toy_df2.add(toy_df1.iloc[:,1],axis='index')
```

	a	b
0	102	202
1	304	404
2	506	606

5.6.3 Case study

To see the application of arithematic operations on DataFrames, let us see the case study below.

Song recommendation: Spotify recommends songs based on songs listened by the user. Suppose you have listened 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 the available song information 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 several 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 similarity 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	2
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 distribution: $U[0,1]$.

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)
```

The above two blocks of code can be concisely written with the `lambda` function 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.*

```
#Index of the row consisting of drivers license can be found with the index attribute
drivers_license_index = spotify_data[spotify_data.track_name=='drivers license'].index[0]
```

Note that the object returned by the `index` attribute is of type `pandas.core.indexes.numeric.Int64Index`. The elements of this object can be retrieved like the elements of a python list. That is why the object is sliced with `[0]` to return the first element of the object. As there is only one observation with the `track_name` as *drivers license*, we sliced the first element. If there were multiple observations with `track_name` as *drivers license*, we will obtain the indices of all those observations with the `index` attribute.

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

```
#Audio features of drivers license are being subtracted from audio features of all songs b
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 similar to *drivers license*!

5.7 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 causality 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
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

	artist_followers	artist_popularity	track_popularity	duration_ms	explicit	releas
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*.

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

5.7.1 Practice exercise 2

5.7.1.1

Use the updated dataset from Practice exercise 1.

The last four variables in the dataset are:

1. `cant_change_math_ability`
2. `can_change_math_ability`
3. `math_is_genetic`
4. `much_effort_is_lack_of_talent`

Each of the above variables has values - `Agree` / `Disagree`. Replace `Agree` with `1` and `Disagree` with `0`.

Hint : You can do it with any one of the following methods:

1. Use the map() function
2. Use the apply() function with the lambda function
3. Use the replace() function
4. Use the applymap() function

Two of the above methods avoid a `for`-loop. Which ones?

Solution:

```
#Making a copy of data to avoid changing the original data.
survey_data_copy = survey_data.copy()

#Using the map function
for i in range(47,51):
    survey_data_copy.iloc[:,i] = survey_data_copy.iloc[:,i].map({'Agree':1,'Disagree':0})
survey_data_copy.iloc[:,47:51].head()
```

	cant_change_math_ability	can_change_math_ability	math_is_genetic	much_effort_is_lack_of_ta
0	0	1	0	0
1	0	1	0	0
2	0	1	0	0
3	0	1	0	0
4	1	0	0	0

```
#Making a copy of data to avoid changing the original data.
survey_data_copy = survey_data.copy()

#Using the lambda function with apply()
```

```
for i in range(47,51):
    survey_data_copy.iloc[:,i] = survey_data_copy.iloc[:,i].apply(lambda x: 1 if x=='Agree
survey_data_copy.iloc[:,47:51].head()
```

	cant_change_math_ability	can_change_math_ability	math_is_genetic	much_effort_is_lack_of_ta
0	0	1	0	0
1	0	1	0	0
2	0	1	0	0
3	0	1	0	0
4	1	0	0	0

```
#Making a copy of data to avoid changing the original data.
survey_data_copy = survey_data.copy()

#Using the replace() function
survey_data_copy.iloc[:,47:51] = survey_data_copy.iloc[:,47:51].replace('Agree','1')
survey_data_copy.iloc[:,47:51] = survey_data_copy.iloc[:,47:51].replace('Disagree','0')
survey_data_copy.iloc[:,47:51].head()
```

	cant_change_math_ability	can_change_math_ability	math_is_genetic	much_effort_is_lack_of_ta
0	0	1	0	0
1	0	1	0	0
2	0	1	0	0
3	0	1	0	0
4	1	0	0	0

```
#Making a copy of data to avoid changing the original data.
survey_data_copy = survey_data.copy()

#Using the lambda function with applymap()
survey_data_copy.iloc[:,47:51] = survey_data_copy.iloc[:,47:51].applymap(lambda x: 1 if x=
survey_data_copy.iloc[:,47:51].head()
```

	cant_change_math_ability	can_change_math_ability	math_is_genetic	much_effort_is_lack_of_ta
0	0	1	0	0
1	0	1	0	0
2	0	1	0	0

	cant_change_math_ability	can_change_math_ability	math_is_genetic	much_effort_is_lack_of_ta
3	0	1	0	0
4	1	0	0	0

5.7.1.2

Among the four variables, which one is the most negatively correlated with `math_is_genetic`?

```
#Computing correlation
survey_data_copy.iloc[:,47:51].corrwith(survey_data_copy.math_is_genetic)
```

```
cant_change_math_ability         0.294544
can_change_math_ability         -0.361546
math_is_genetic                  1.000000
much_effort_is_lack_of_talent    0.154083
dtype: float64
```

The variable `can_change_math_ability` is the most negatively correlated wtih `math_is_genetic`.

6 Data visualization

<IPython.core.display.Image object>

"One picture is worth a thousand words" - Fred R. Barnard

Visual perception offers the highest bandwidth channel, as we acquire much more information through visual perception than with all of the other channels combined, as billions of our neurons are dedicated to this task. Moreover, the processing of visual information is, at its first stages, a highly parallel process. Thus, 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 irregularities 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 visualizations.

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.

6.1 Matplotlib

Matplotlib is:

- a low-level graph plotting library in python that strives to emulate MATLAB,
- can be used in Python scripts, Python and IPython shells, Jupyter notebooks and web application servers.
- is mostly written in python, a few segments are written in C, Objective-C and Javascript for Platform compatibility.

Conceptual model: Plotting requires action on a range of levels, ranging from the size of the figure to the text object in the plot. Matplotlib provides **object-oriented** interface in the hierarchical fashion to provide complete control over the plot. The user generates and keeps track of the figure and axes objects. These axes objects are then used for most plotting actions.

6.1.1 Matplotlib: Object hierarchy

A *hierarchy* means that there is a tree-like structure of matplotlib objects underlying each plot.

A *Figure* object is the outermost container for a matplotlib graphic, which can contain multiple *Axes* objects. Note that an *Axes* actually translates into what we think of as an individual plot or graph (rather than the plural of *axis* as we might expect).

The *Figure* object is a box-like container holding one or more Axes (actual plots), as shown in Figure 6.1. Below the *Axes* in the hierarchy are smaller objects such as *tick marks, individual lines, legends, and text boxes*. Almost every *element* of a chart is its own manipulable Python object, all the way down to the *ticks* and *labels*.

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

Figure 6.1: Matplotlib Object hierarchy

However, Matplotlib presents this as a figure anatomy, rather than an explicit hierarchy. Figure 6.2 shows the components of a figure that can be customized with Matplotlib. *(Source: https://matplotlib.org/stable/gallery/showcase/anatomy.html).*

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

Figure 6.2: Matplotlib anatomy of a figure

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

6.1.2 Scatterplots and trendline with Matplotlib

Purpose of scatterplots: Scatterplots (with or without a trendline) allow us to visualize the relationship between two numerical variables.

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 labeling the plot axes.

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

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

There are two ways of plotting the figure:

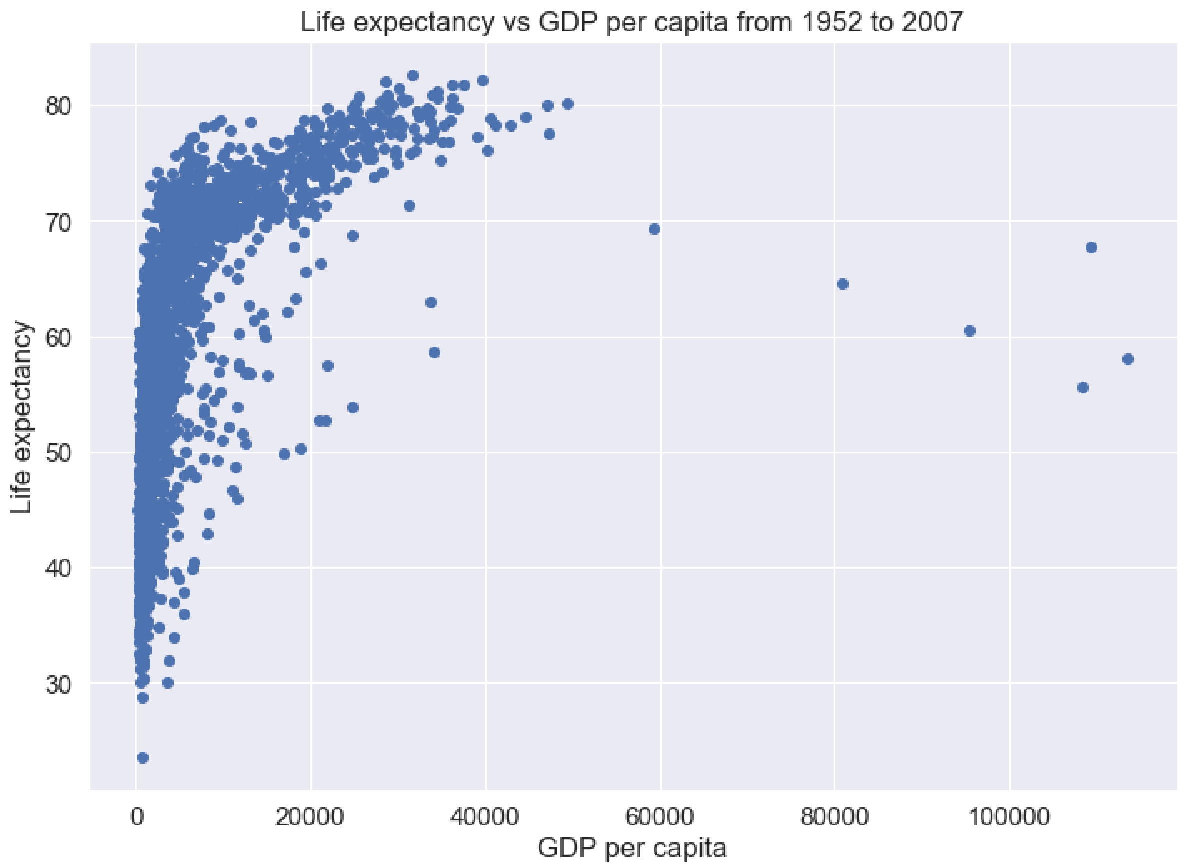
1. Explicitly creating figures and axes, and call methods on them *(object-oriented style)*.
2. Letting pyplot implicitly track the plot that it wants to reference. Simple functions are used to add plot elements (lines, images, text, etc.) to the current axes in the current figure *(pyplot-style)*.

We'll plot the figure in both ways.

```
#Method 1: Object-oriented style
fig, ax = plt.subplots() #Create a figure and an axes
x = gdp_data.gdpPercap
y = gdp_data.lifeExp
ax.plot(x,y,'o')   #Plot data on the axes
ax.set_xlabel('GDP per capita')    #Add an x-label to the axes
```

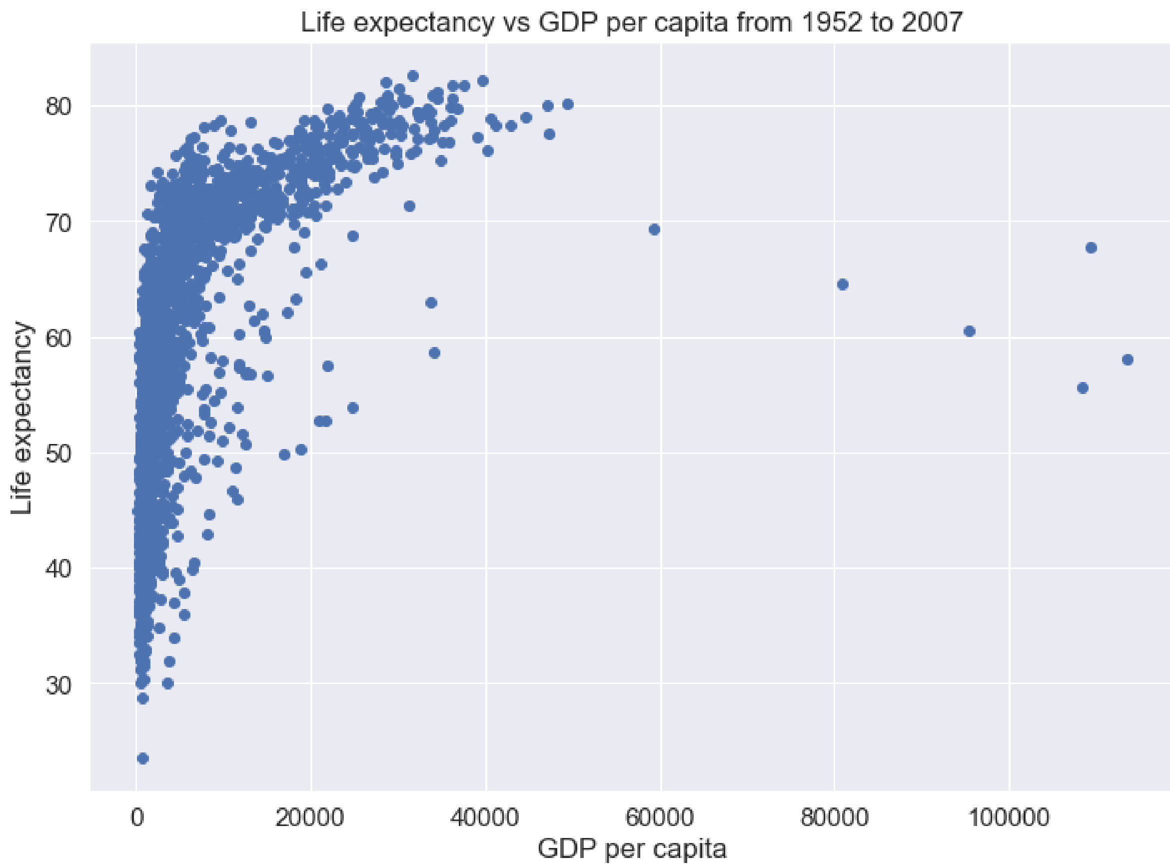
```
ax.set_ylabel('Life expectancy')   #Add a y-label to the axes
ax.set_title('Life expectancy vs GDP per capita from 1952 to 2007')
```

```
Text(0.5, 1.0, 'Life expectancy vs GDP per capita from 1952 to 2007')
```




```
#Method 2: pyplot style
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
plt.title('Life expectancy vs GDP per capita from 1952 to 2007')
```

```
Text(0.5, 1.0, 'Life expectancy vs GDP per capita from 1952 to 2007')
```



Both the plotting styles - object-oriented style and the pyplot style are perfectly valid and have their pros and cons.

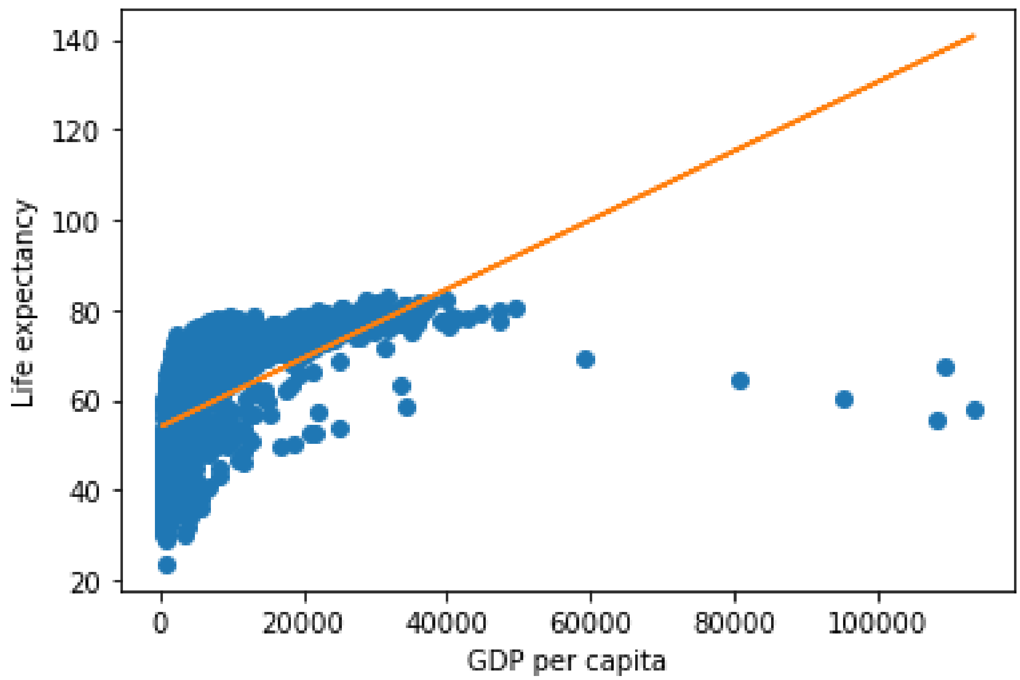
- Pyplot style is easier for simple plots
- Object-oriented style is slightly more complicated but more powerful as it allows for greater control over the axes in figure. This proves to be quite useful when we are dealing with a figure with multiple axes.

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 in 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.

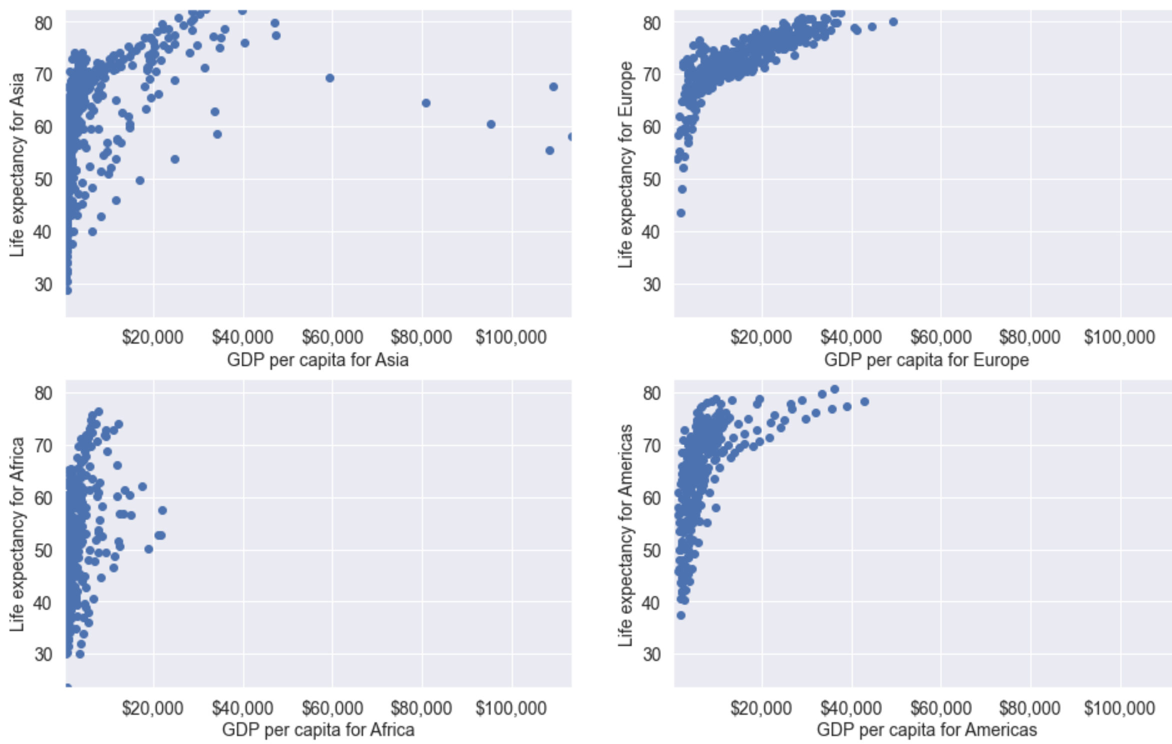
We used the NumPy function polyfit() to compute the slope and intercept of the trendline. Then, we defined an object `compute_y_given_x` of poly1d class and used it to compute the trendline.

6.1.3 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=(16,10))
plt.subplots_adjust(wspace=0.2) #adjusting white space between individual plots

#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):

        #Getting the GDP per capita and life expectancy of the countries of the (i,j)th co
        x = gdp_data.loc[gdp_data.continent==continents[i,j],:].gdpPercap
        y = gdp_data.loc[gdp_data.continent==continents[i,j],:].lifeExp

        #Making the scatterplot
        axes[i,j].plot(x,y,'o')

        #Setting limits on the 'x' and 'y' axes
        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()])

        #Labelling the 'x' and 'y' axes
        axes[i,j].set_xlabel('GDP per capita for '+ continents[i,j],fontsize = 14)
        axes[i,j].set_ylabel('Life expectancy for '+ continents[i,j],fontsize = 14)

        #Putting a dollar sign, and thousand-comma separator on x-axis labels
        axes[i,j].xaxis.set_major_formatter('${x:,.0f}')

        #Increasing font size of axis labels
        axes[i,j].tick_params(axis = 'both',labelsize=14)
```



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.

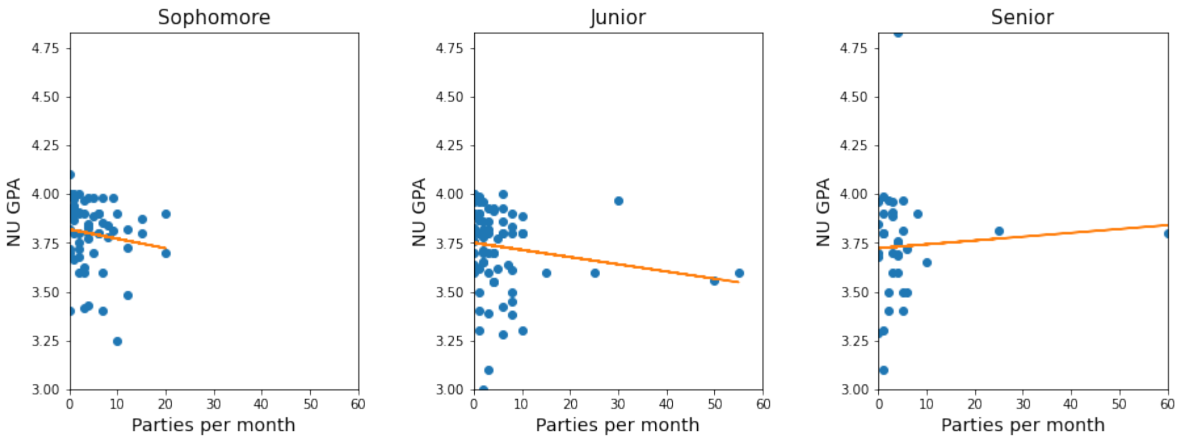
6.1.4 Practice problem 1

Is `NU_GPA` associated with `parties_per_month`? Analyze the association separately for *Sophomores, Juniors, and Seniors* (categories of the variable `school_year`).

Make scatterplots of `NU_GPA` vs `parties_per_month` in a 1 x 3 grid, where each grid is for a distinct `school_year`. Plot the trendline as well for each scatterplot. Use the file *survey_data_clean.csv*.

Solution:

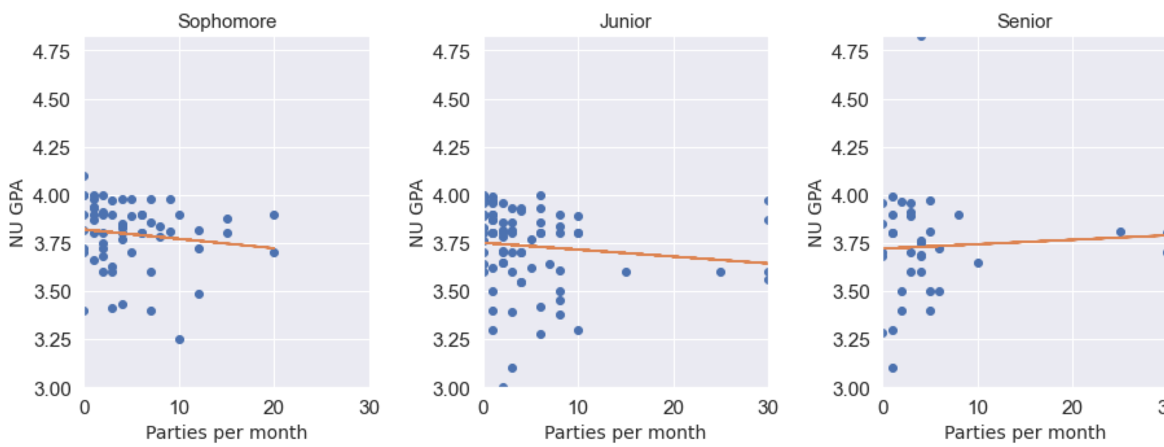
```
survey_data = pd.read_csv('./Datasets/survey_data_clean.csv')

def NU_GPA_vs_parties_per_month(data):
    fig, axes = plt.subplots(1,3,figsize=(15,5))
    plt.subplots_adjust(wspace=0.4)

    school_years = np.array(['Sophomore', 'Junior','Senior'])
    for i in range(3):
        x = data.loc[data.school_year==school_years[i],:].parties_per_month
        y = data.loc[data.school_year==school_years[i],:].NU_GPA

        #The data has missing values. We can draw a trendline using only the non-missing v
        #`idx_non_missing` will have the indices of the non-missing value-pairs of NU_GPA
        idx_non_missing = np.isfinite(x) & np.isfinite(y)

        axes[i].plot(x,y,'o',label = school_years[i])
        axes[i].set_xlim([data.parties_per_month.min(), data.parties_per_month.max()])
        axes[i].set_ylim([data.NU_GPA.min(), data.NU_GPA.max()])
        axes[i].set_xlabel('Parties per month',fontsize = 14)
        axes[i].set_ylabel('NU GPA',fontsize = 14)
        axes[i].set_title(school_years[i],fontsize = 15)
        slope_intercept_trendline = np.polyfit(x[idx_non_missing],y[idx_non_missing],1)
        compute_y_given_x = np.poly1d(slope_intercept_trendline) #Defining a function that
        axes[i].plot(x,compute_y_given_x(x)) #Plotting the trendline

NU_GPA_vs_parties_per_month(survey_data)
```



Note that the trendline in the above plots seems to be influenced by a few points having extreme values of `parties_per_month`. These points have a *high leverage* (a concept we'll learn in a future course on linear regression) in influencing the trendline. So, we should visualize the trend by removing or capping these *high-leverage* points, to avoid the distortion of the trend by a few points.

Let us cap the the values of `parties_per_month` to 30, and make the visualizations again.

```
survey_data_parties_capped = survey_data.copy()
survey_data_parties_capped.parties_per_month = survey_data.parties_per_month.apply(lambda
NU_GPA_vs_parties_per_month(survey_data_parties_capped)
```



We see that the trend didn't change much after removing the *high leverage* points. *(Note that although the high leverage points have the leverage to influence the trendline, they need not necessarily influence it)*. From the visualization, `NU_GPA` doesn't seem to be associated with `parties_per_month` for students of any of the school years.

6.1.5 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']])
plt.rcParams["figure.figsize"] = (9,6)
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 0x2320d6d70a0>
```

Note that a disadvantage of the above plot is overplotting. The data points corresponding to the *Americas* are hiding the data points of other continents. However, if the data points corresponding to different categories are spread apart, then it may be convenient to visualize all the categories on the same plot.

6.2 Pandas

Matplotlib is a 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.

6.2.1 Scatterplots with Pandas

```
#Plotting life expectancy vs GDP per capita using the Pandas plot() function
ax = gdp_data.plot(x = 'gdpPercap', y = 'lifeExp', kind = 'scatter',figsize=(10, 6),xlabel
               ylabel = 'Life expectancy')
ax.xaxis.set_major_formatter('${x:,.0f}')
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as val
```

In the above plot, note that:

- With matplotlib, it will take 3 lines to make the same plot - one for the scatterplot, and two for the axis titles.
- The object `ax` is of type *matplotlib.axes._subplots.AxesSubplot (check the code below)*. This means we can use the attributes and methods associated with the `axes` object of Matplotlib. If you see the documentation of the Pandas `plot()` function, you will find that under the `kwargs**` argument, you have *Options to pass to matplotlib plotting method.* Thus, you get the convenience of using the Pandas `plot()` function, while also having the attributes and methods associated with Matplotlib.

```
type(ax)
```

```
matplotlib.axes._subplots.AxesSubplot
```

6.2.2 Lineplots with Pandas

Purpose of lineplots: Lineplots show the relationship between two numerical variables when the variable on the x-axis, also called the *explanatory variable*, is of a sequential nature; in other words there is an inherent ordering to the variable. The most common example of lineplots have some notion of time on the x-axis (or the horizontal axis): hours, days, weeks, years, etc. Since time is sequential, we connect consecutive observations of the variable on the y-axis with a line. Lineplots that have some notion of time on the x-axis are also called time series plots. Lineplots should be avoided when there is not a clear sequential ordering to the variable on the x-axis.

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.

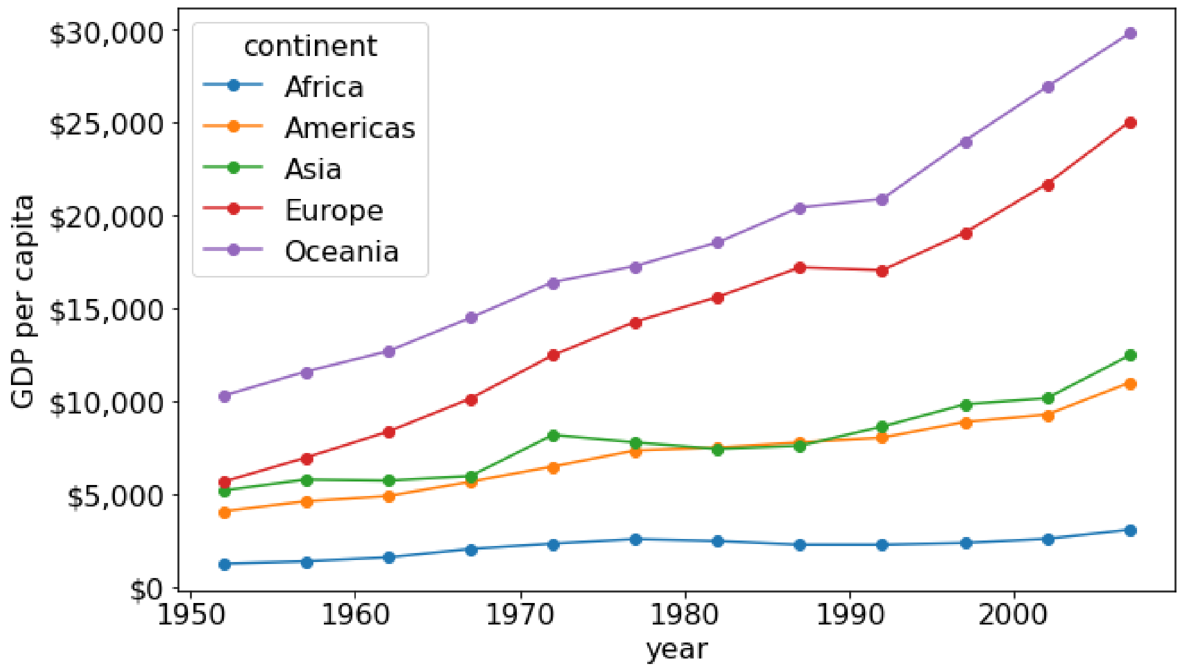
```
#You may ignore this code block until Chapter 8.
mean_gdp_per_capita = gdp_data.pivot_table(index = 'year', columns = 'continent',values =
mean_gdp_per_capita.head()
```

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.

```
ax = mean_gdp_per_capita.plot(ylabel = 'GDP per capita',figsize = (10,6),marker='o')
ax.yaxis.set_major_formatter('${x:,.0f}')
```



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

Note that argument *marker = 'o'* puts a solid circle at each of the data points.

6.2.3 Bar plots with Pandas

Purpose of bar plots: Barplots are used to visualize any aggregate statistics of a continuous variable with respect to the categories or levels of a categorical variable. For example, we may visualize the average IMDB rating *(aggregate statistics)* of movies based on their genre *(the categorical variable)*.

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

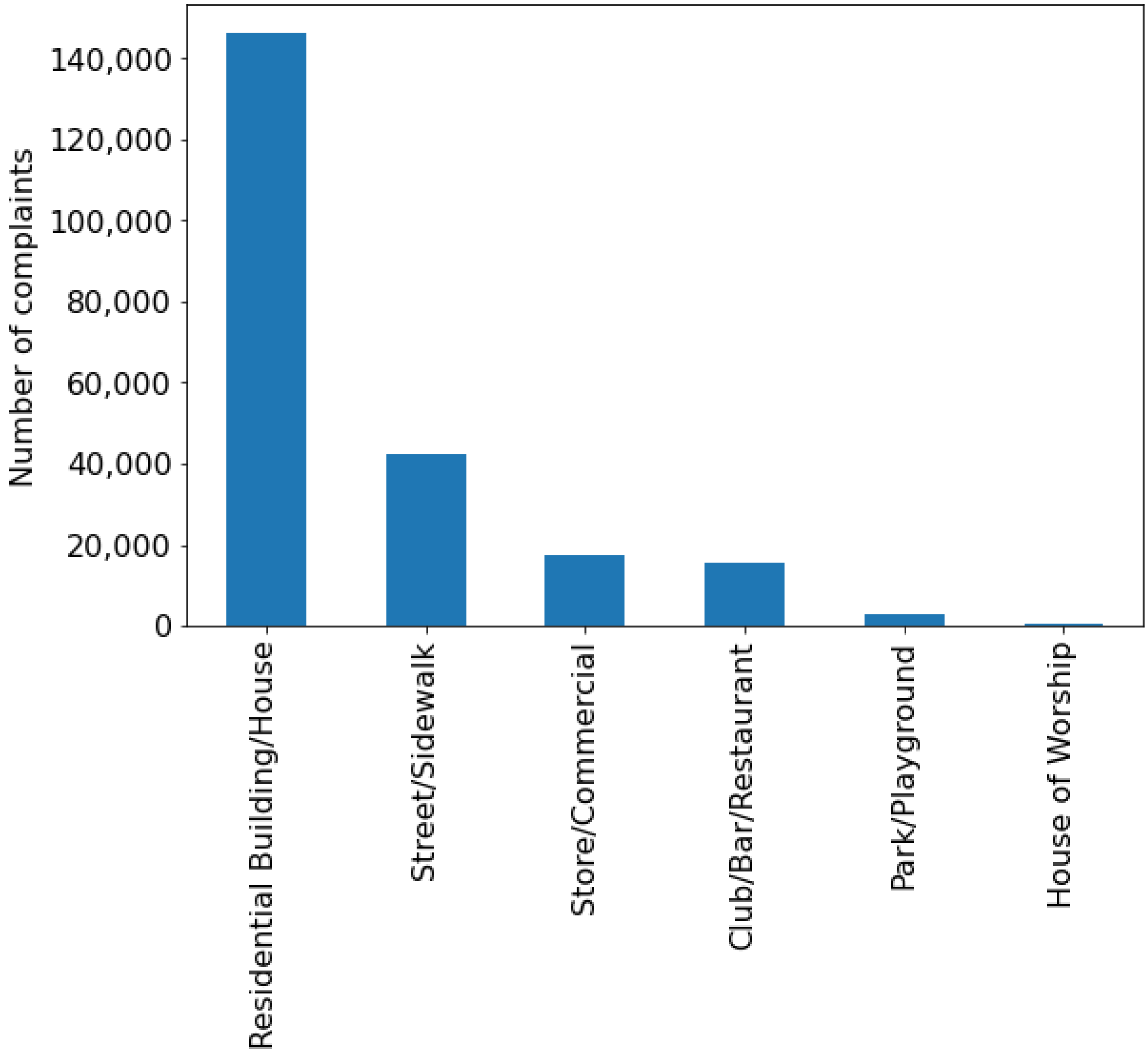
Below, we are reading the dataset of noise complaints of type *Loud music/Party* received the police in New York City in 2016.

```
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	MANHAT
1	12/31/2015 0:02	12/31/2015 4:36	Store/Commercial	10040.0	NEW YORK	MANHAT
2	12/31/2015 0:03	12/31/2015 0:40	Residential Building/House	10026.0	NEW YORK	MANHAT
3	12/31/2015 0:03	12/31/2015 1:53	Residential Building/House	11231.0	BROOKLYN	BROOKI
4	12/31/2015 0:05	12/31/2015 3:49	Residential Building/House	10033.0	NEW YORK	MANHAT

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

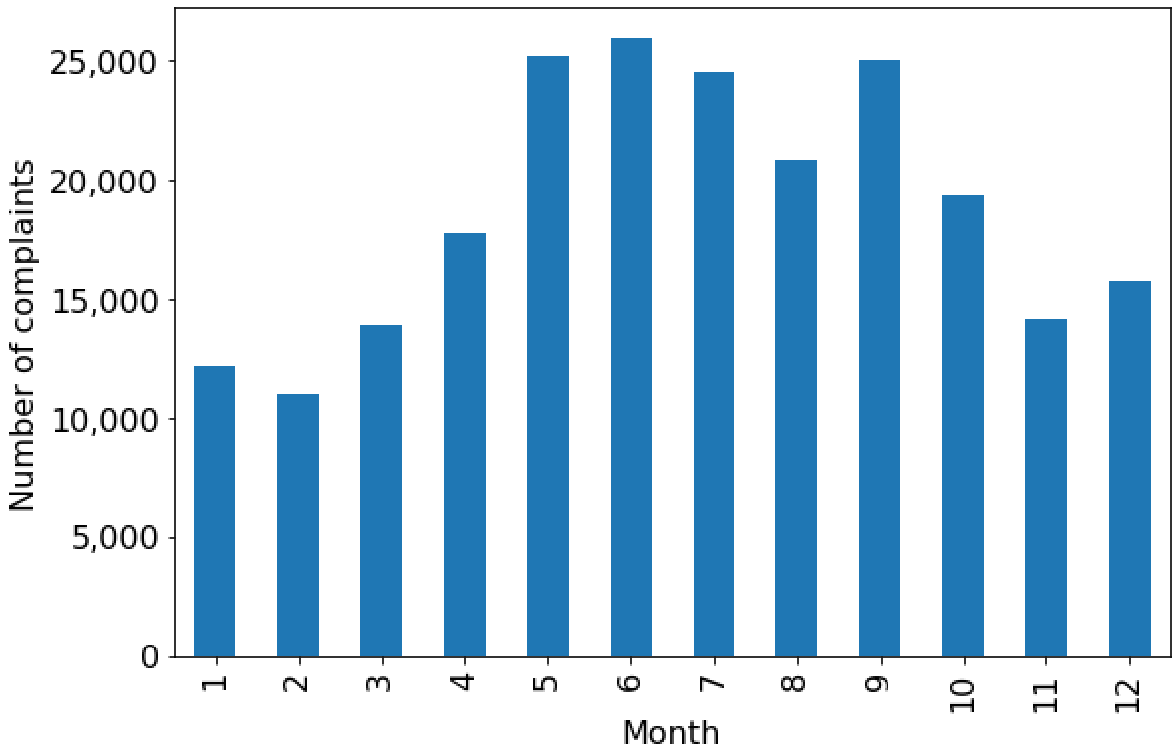
```
#Using the pandas function bar() to create bar plot
ax = nyc_party_complaints['Location Type'].value_counts().plot.bar(ylabel = 'Number of com
ax.yaxis.set_major_formatter('{x:,.0f}')
```



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
ax = nyc_party_complaints['Month_of_the_year'].value_counts().sort_index().plot.bar(ylabel
                                                                                   xlabel = "Mo
ax.yaxis.set_major_formatter('{x:,.0f}')
```



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
ax = complaints_location.plot.bar(stacked=True,ylabel = 'Number of complaints',figsize=(15
ax.tick_params(axis = 'both',labelsize=15)
ax.yaxis.set_major_formatter('{x:,.0f}')
```

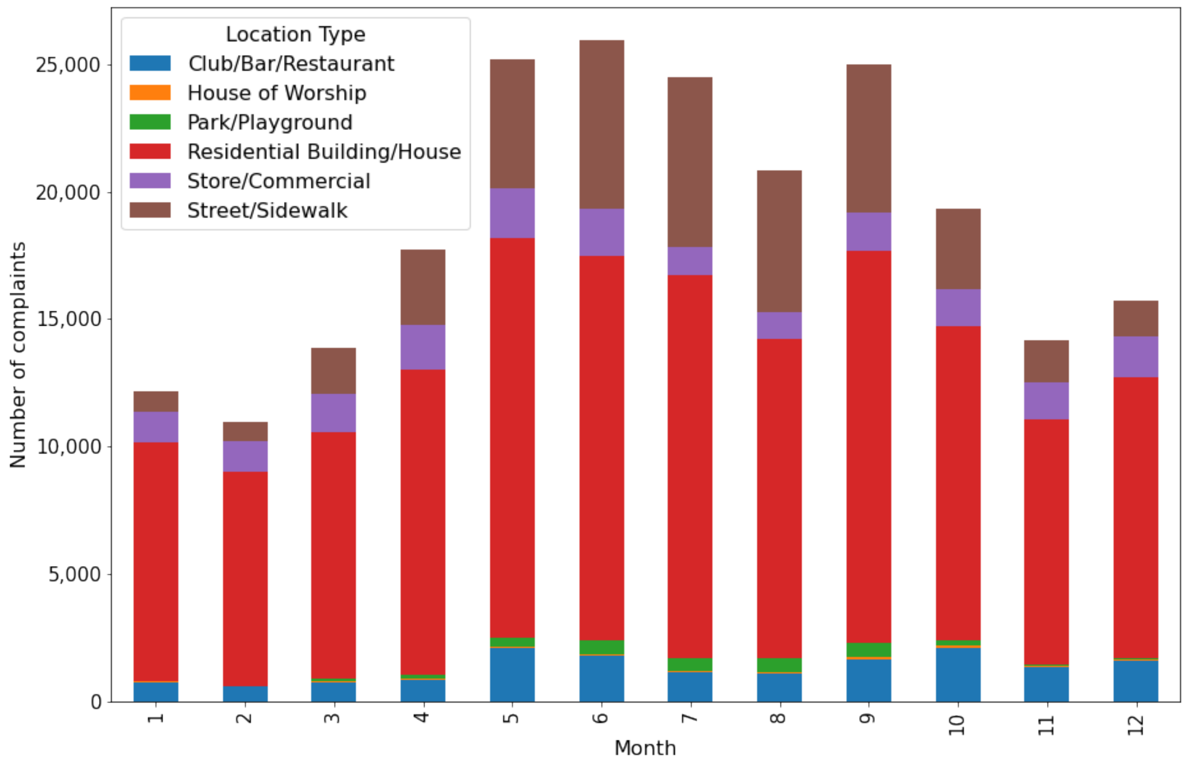
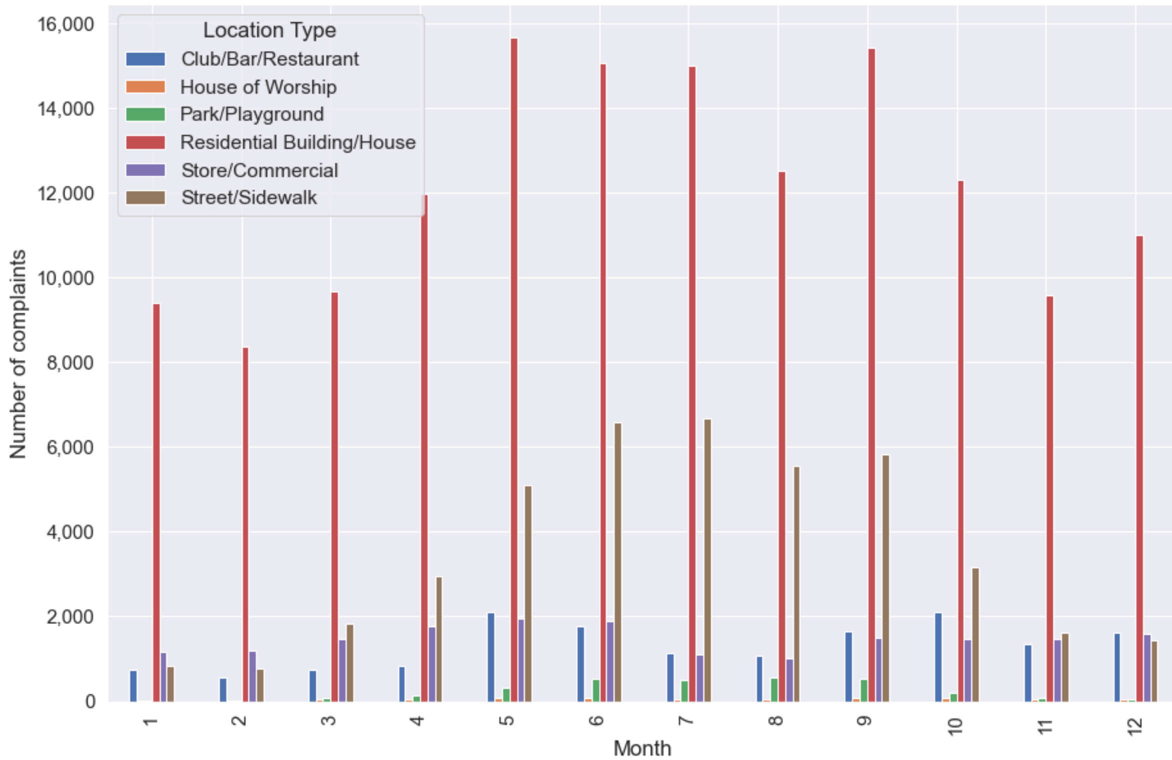


Figure 6.3: Stacked bar plot with Pandas

The above plots gives the insights about location and day of the year simultaneously that were previously separately obtained by the individual plots.

An alternative to stacked barplots are *side-by-side* barplots, as shown below.

```
#Side-by-side bar plot showing number of complaints at different months of the year, and f
ax = complaints_location.plot.bar(ylabel = 'Number of complaints',figsize=(15, 10), xlabel
ax.tick_params(axis = 'both',labelsize=15)
ax.yaxis.set_major_formatter('{x:,.0f}')
```



Q1 In which scenarios should we use a stacked barplot instead of a side-by-side barplot and vice-versa?

6.3 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
```

6.3.1 Bar plots with confidence intervals with Seaborn

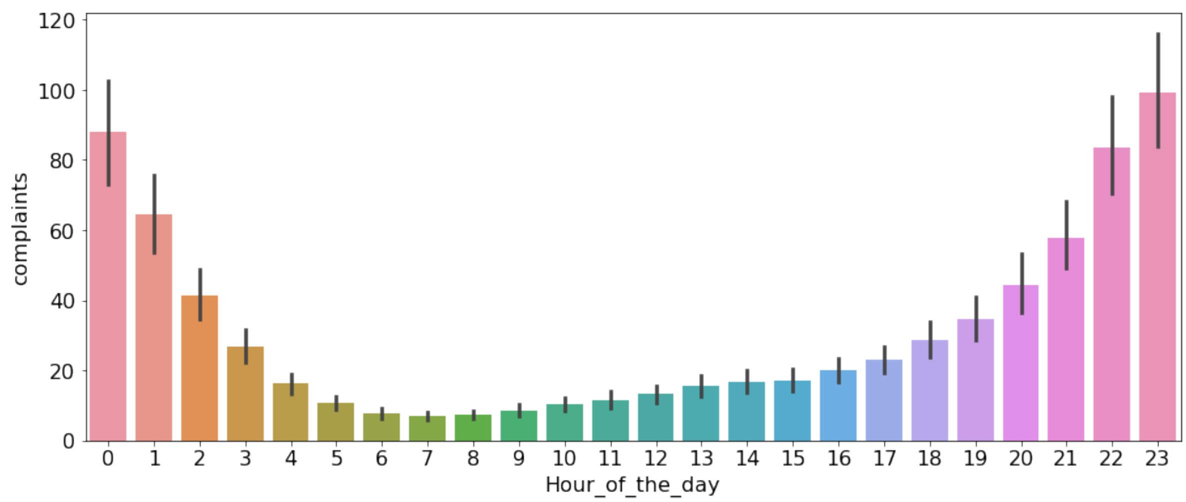
We'll group the data to obtain the total complaints for each *Location Type, Borough, Month_of_the_year, and Hour_of_the_day.* Note that you'll learn grouping data in Chapter 9, so you may ignore the next code block. The grouping is done to shape the data in a suitable form for visualization.

```
#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.

```
ax = sns.barplot(x="Hour_of_the_day", y = 'complaints',  data=nyc_complaints_grouped)
ax.figure.set_figwidth(15)
```



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.

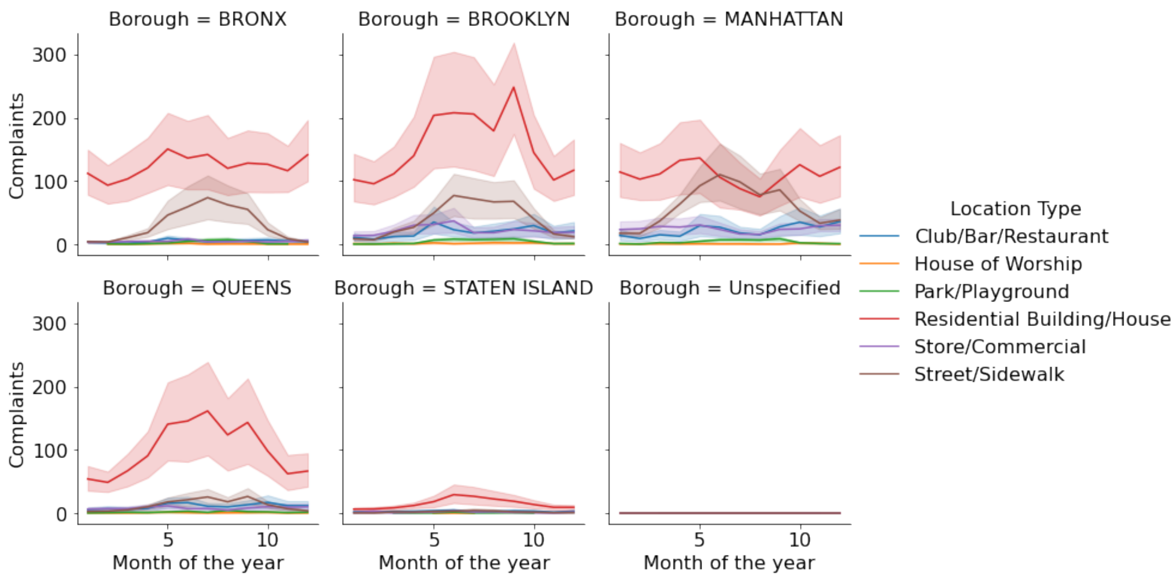
6.3.2 Facetgrid: Multi-plot grid for plotting conditional relationships

With pandas, we simultaneously visualized the number of complaints with month of the year and location type in Figure 6.3. We'll use Seaborn to add another variable - Borough to the visualization.

Q: Visualize the mean 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.set_axis_labels("Month of the year", "Complaints")
a.add_legend()
```



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 Sydney, Canberra, and Melbourne from 2007 to 2017.

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

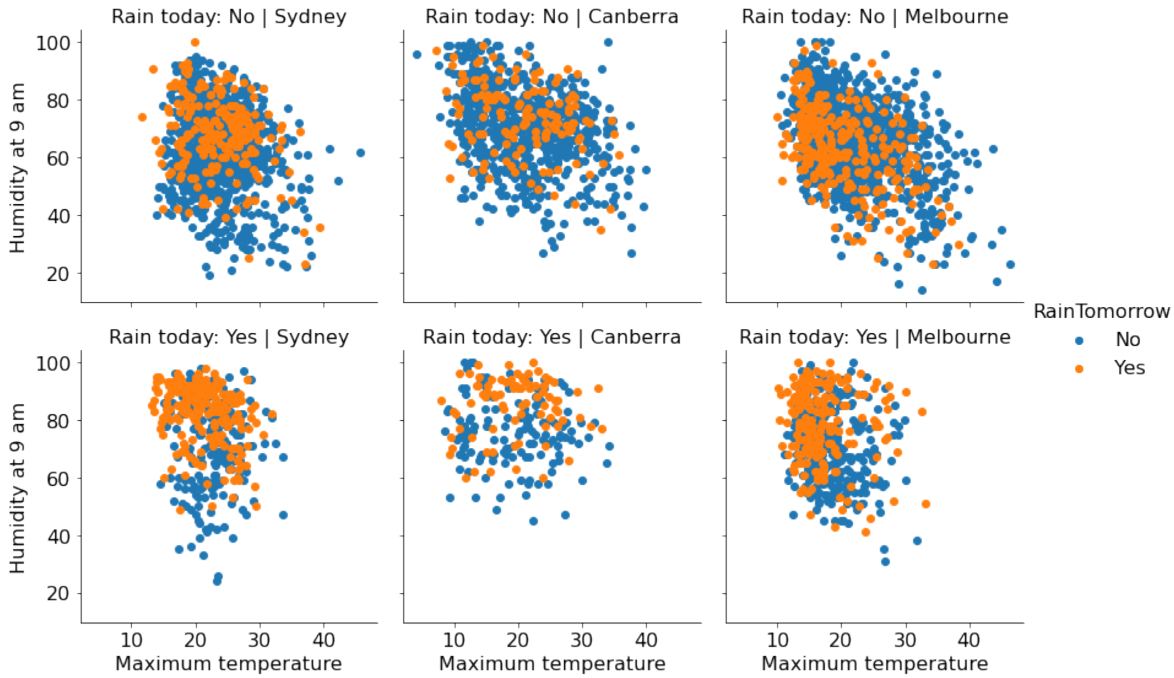
	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	Win
0	10/20/2010	Sydney	12.9	20.3	0.2	3.0	10.9	ENE	37
1	10/21/2010	Sydney	13.3	21.5	0.0	6.6	11.0	ENE	41
2	10/22/2010	Sydney	15.3	23.0	0.0	5.6	11.0	NNE	41
3	10/26/2010	Sydney	12.9	26.7	0.2	3.8	12.1	NE	33
4	10/27/2010	Sydney	14.8	23.8	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 = 1,hue
a.map(plt.scatter,'MaxTemp','Humidity9am')
a.set_axis_labels("Maximum temperature", "Humidity at 9 am")
a.set_titles(col_template="{col_name}", row_template="Rain today: {row_name}")
a.add_legend()
```



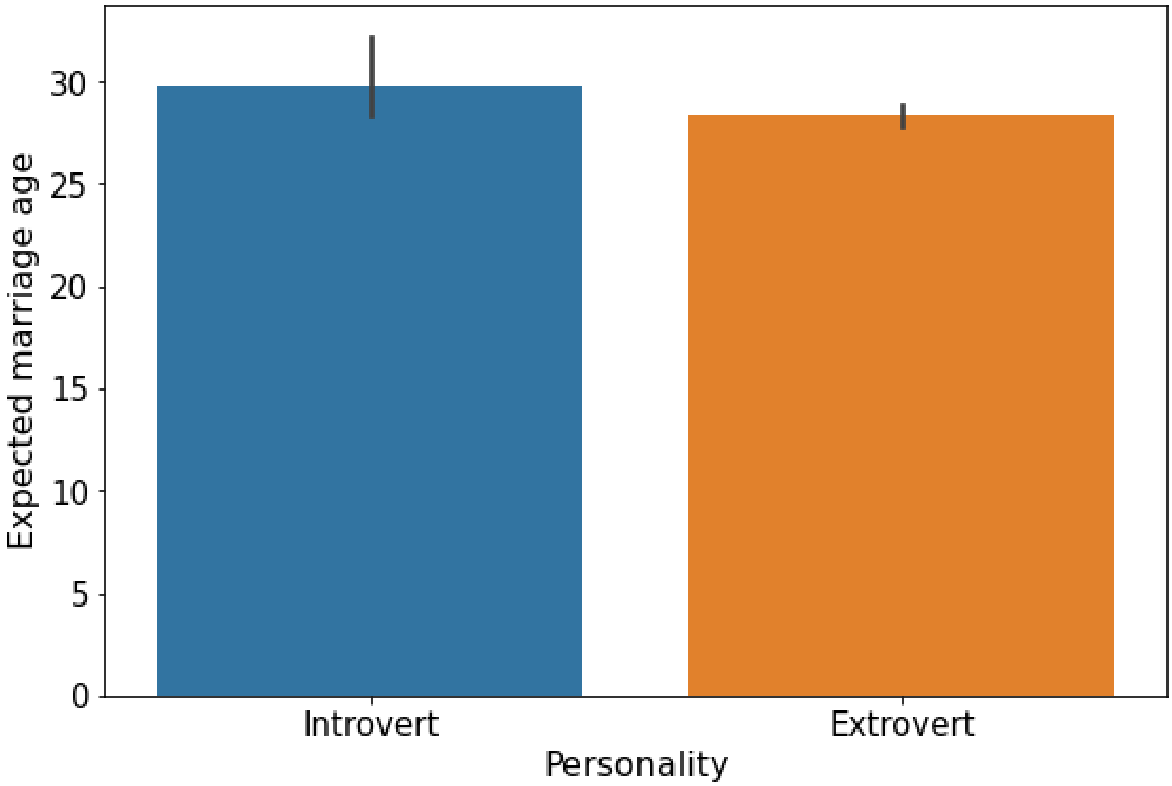
Humidity tends to be higher when it is going to rain the next day. However, the correlation is much more pronounced for Syndey. In case it is not raining on the current day, humidity seems to be slightly negatively correlated with temperature.

6.3.3 Practice exercise 2

How does the expected marriage age of the people of STAT303-1 depend on their characteristics? We'll use visualizations to answer this question. Use data from the file *survey_data_clean.csv*. Proceed as follows:

1. Make a visualization that compares the mean `expected_marriage_age` of introverts and extroverts *(use the variable `introvert_extrovert`)*. What insights do you obtain?

```
plt.rcParams["figure.figsize"] = (9,6)
sns.barplot(x = 'introvert_extrovert' ,y = 'expected_marriage_age', data = survey_data)
plt.xlabel('Personality', fontsize=16);
plt.ylabel('Expected marriage age', fontsize=16);
plt.xticks(fontsize=15);
plt.yticks(fontsize=15);
```



The mean expected marriage age for introverts is about 2 years higher than that for extroverts. Also, there is a higher variation in the expected marriage age of introverts as compared to extroverts.

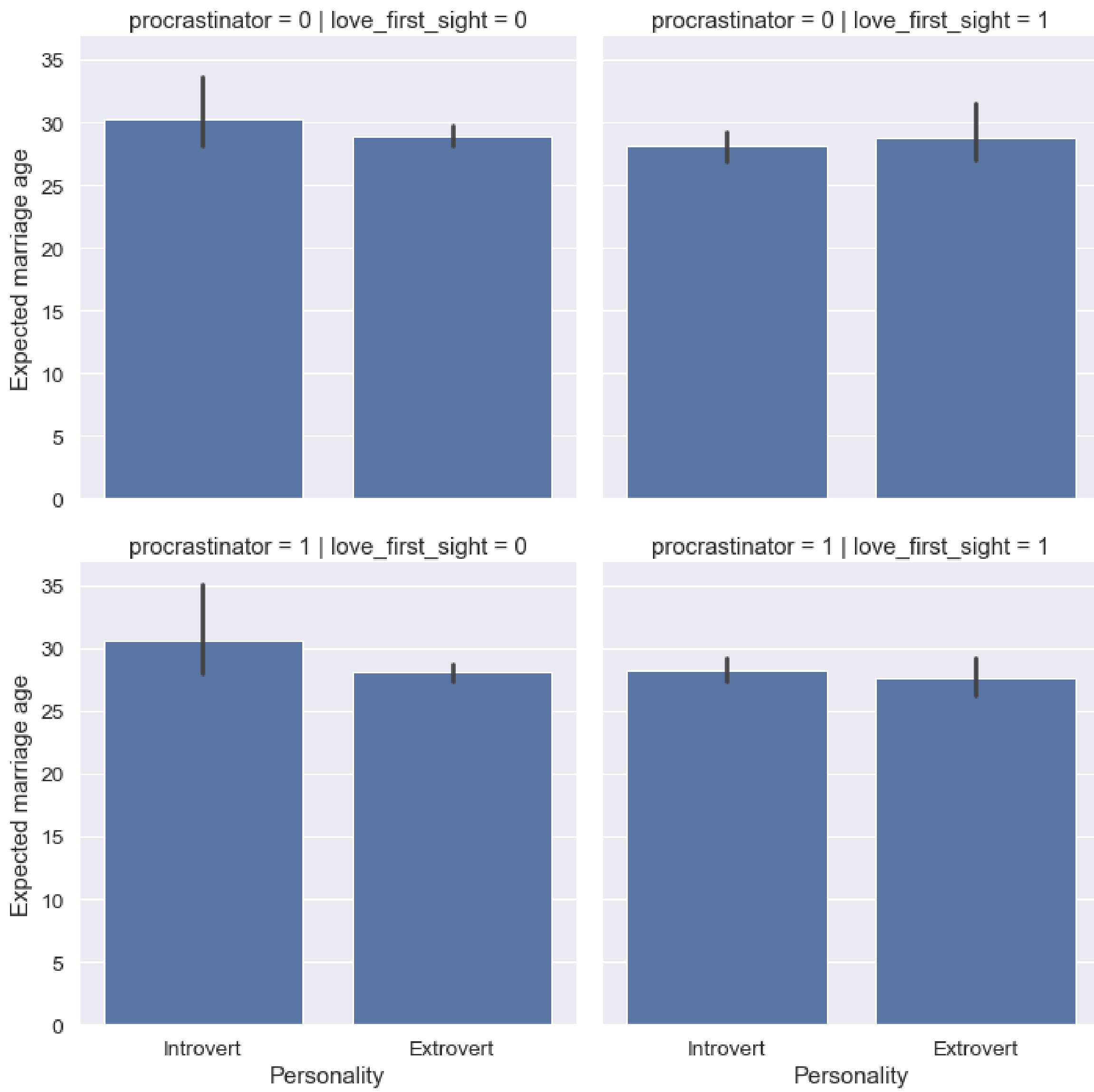
2. Does the mean `expected_marriage_age` of introverts and extroverts depend on whether they believe in love in first sight *(variable name: `love_first_sight`)*? Update the previous visualization to answer the question.

```
sns.set(font_scale=1.25)
a = sns.FacetGrid(survey_data,col='love_first_sight',height = 5,aspect = 1)
a.map(sns.barplot, 'introvert_extrovert' , 'expected_marriage_age' ,order = ['Introvert',
a.set_axis_labels("Personality", "Expected marriage age")
a.add_legend()
```

Yes, only those introverts who do not believe in love in first sight have a higher mean value of expected marriage age.

3. In addition to `love_first_sight`, does the mean `expected_marriage_age` of introverts and extroverts depend on whether they are a procrastinator *(variable name: `procrastinator`)*? Update the previous visualization to answer the question.

```
sns.set(font_scale=1.25)
a = sns.FacetGrid(survey_data,col='love_first_sight',height = 5,aspect = 1, row = 'procras
a.map(sns.barplot, 'introvert_extrovert' , 'expected_marriage_age' ,order = ['Introvert',
a.set_axis_labels("Personality", "Expected marriage age")
a.add_legend()
```



Procrastination does not seem to make much of a difference in the expected marriage age. The mean expected marriage age of procrastinating introverts seems to be only a little higher than the non-procrastinating introverts.

4. Is there any critical information missing in the above visualizations that, if revealed, may cast doubts on the patterns observed in them?

Yes, we don't know the number of observations corresponding to each bar of the bar plots. If there are a very few observations in any of the categories, then the trend shown by that category may not be reliable. For example, in the data (`survey_data`), there are only 8 introverts who are not procrastinators and believe in love in first sight, while there are 52 introverts who are procrastinators and do not believe in love in first sight.

If there are more introverts in the class of STAT303-1 who are not procrastinators and believe in love at first sight (may be they didn't fill the survey), then they are under-represented in the sample of people who filled the survey, and the trend observed for them may be less reliable than that for other people.

```
#Code for finding the number of people in each category - you will understand this code la
survey_data[['introvert_extrovert','love_first_sight','procrastinator','Timestamp']].group
                                                                                   'love_fi
```

introvert_extrovert	love_first_sight	procrastinator	Timestamp
Extrovert	0	0	19
		1	35
	1	0	10
		1	15
Introvert	0	0	32
		1	52
	1	0	8
		1	21

6.3.4 Histogram and density plots with Seaborn

Purpose: Histogram and density plots visualize the distribution of a continuous variable.

A histogram plots the number of observations occurring within discrete, evenly spaced bins of a random variable, to visualize the distribution of the variable. It may be considered a special case of a bar plot as bars are used to plot the observation counts.

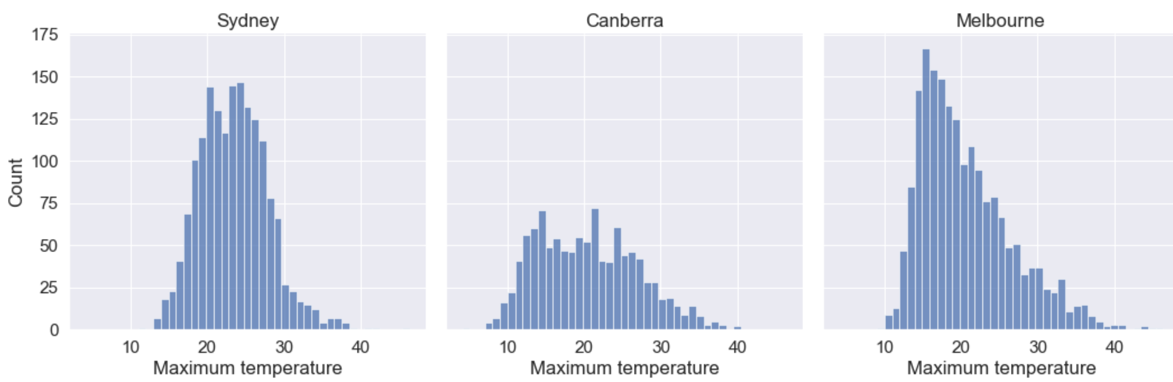
A density plot uses a kernel density estimate to approximate the distribution of random variable.

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.set(font_scale = 1.4)
a = sns.displot(data = aussie_weather, x = 'MaxTemp',kind = 'hist',col='Location')
```

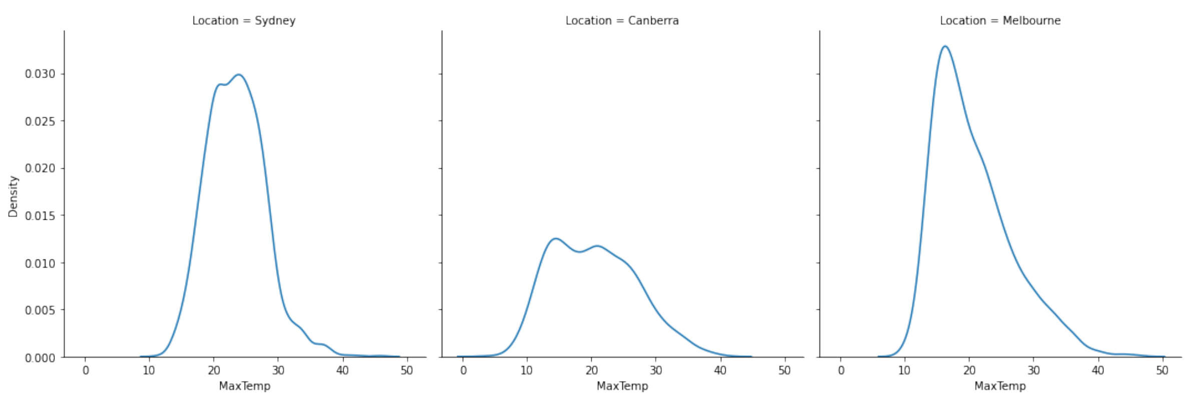
```
a.set_axis_labels("Maximum temperature", "Count")
a.set_titles("{col_name}")
```



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 temperature.

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')
```

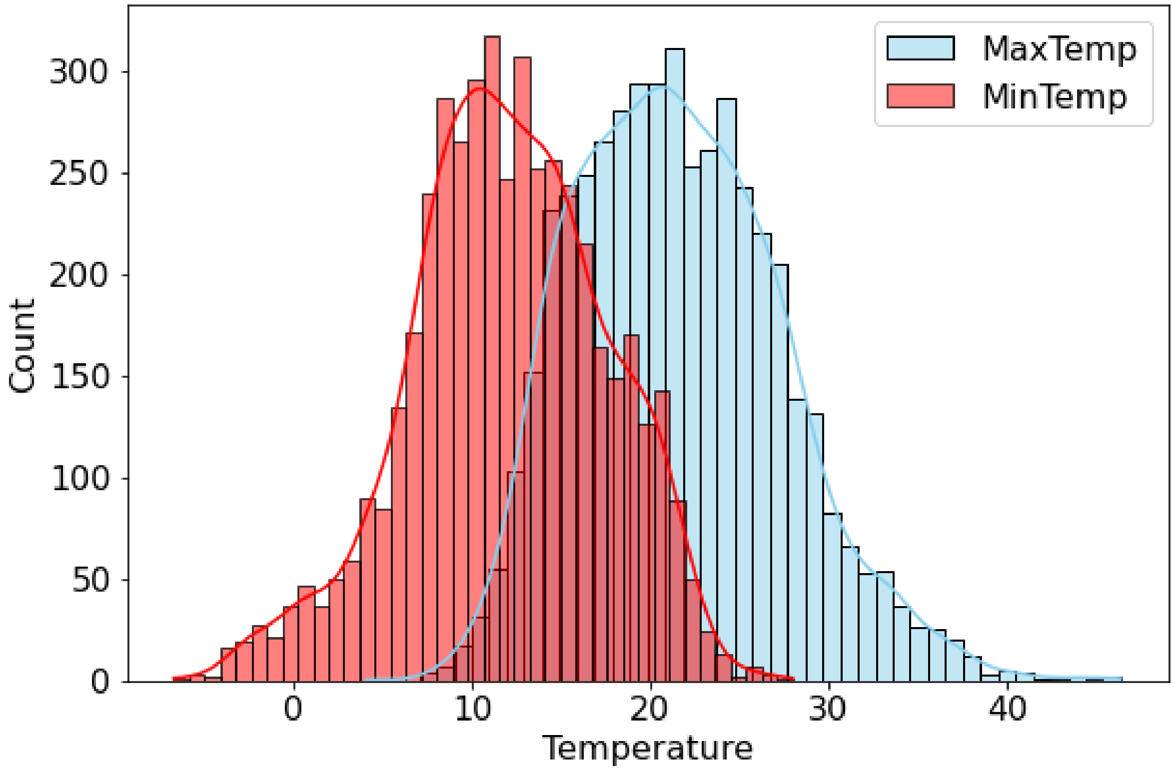


Example: Show the distributions of the maximum and minimum temperatures in a single plot.

```
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()
plt.xlabel('Temperature')
```

```
Text(0.5, 0, 'Temperature')
```



The Seaborn function histplot() can be used to make a density plot overlapping on a histogram.

6.3.5 Boxplots with Seaborn

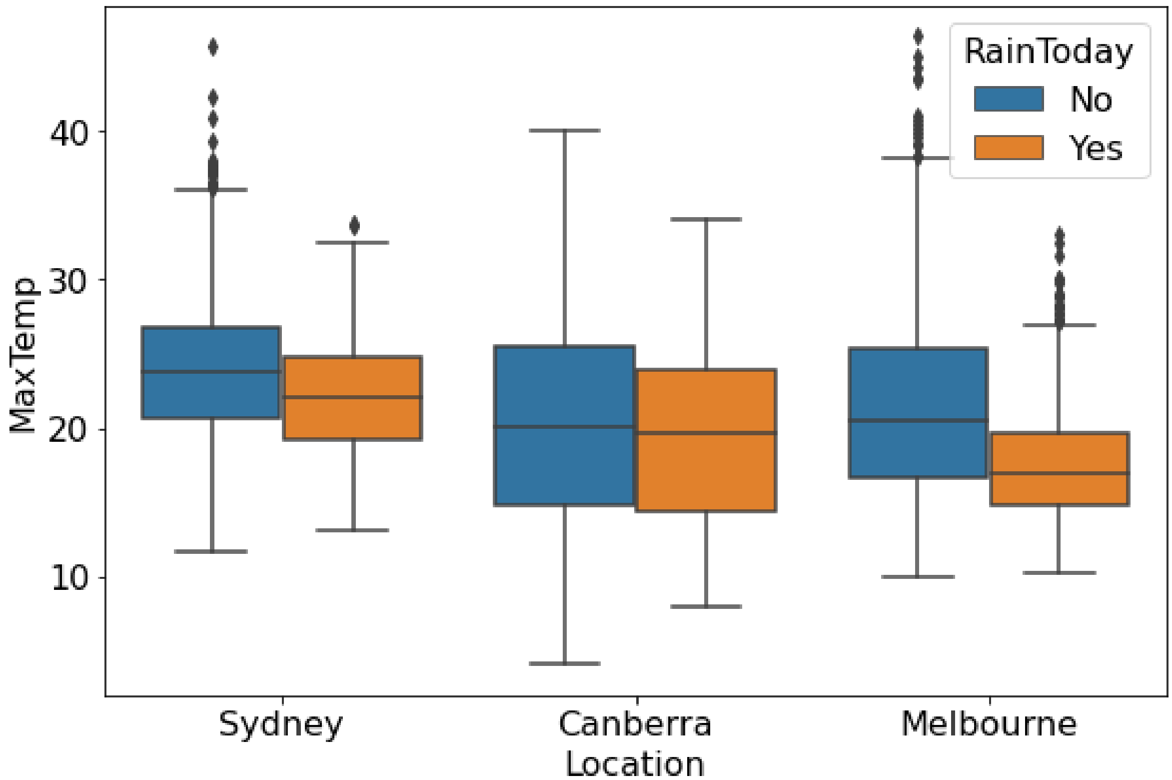
Purpose: Boxplots is a standardized way of visualizing the distribution of a continuous variable. They show five key metrics that describe the data distribution - median, 25th percentile value, 75th percentile value, minimum and maximum, as shown in the figure below. Note that the minimum and maximum 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 Melbourne have some extremely high outlying values of maximum temperature.

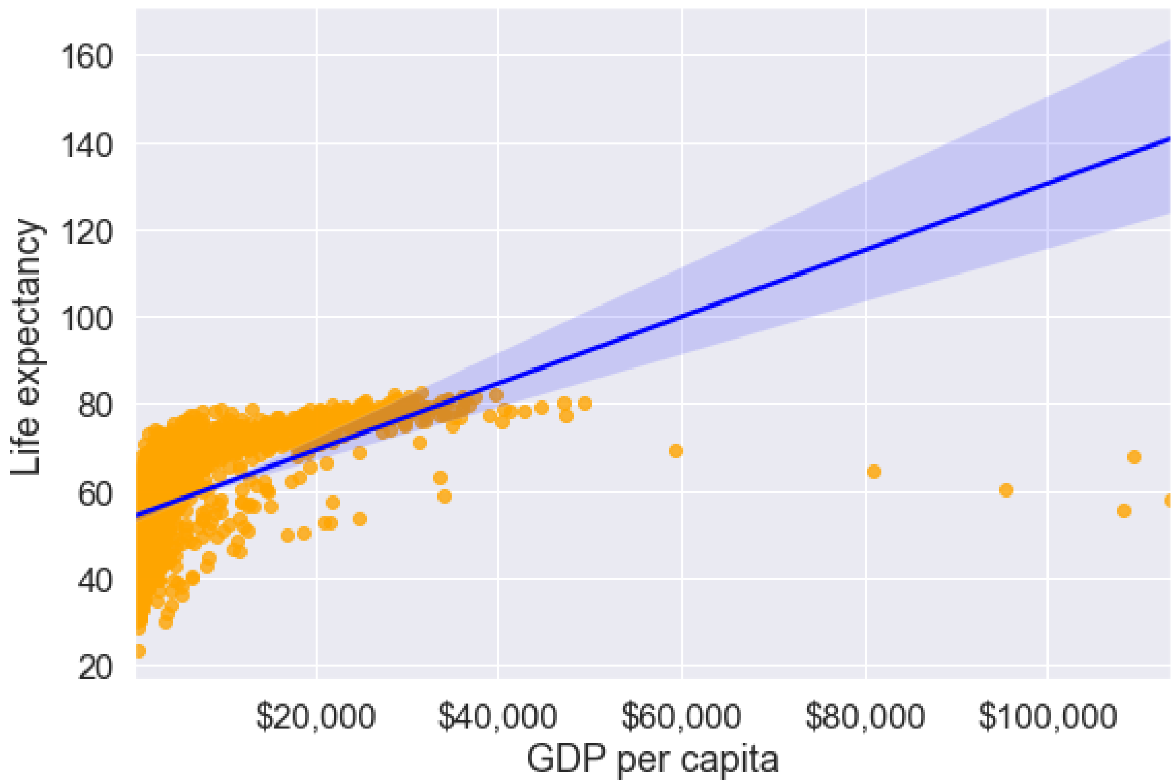
We have used the Seaborn boxplot() function for the above plot.

6.3.6 Scatterplots with Seaborn

We made scatterplots with Matplotlib and Pandas earlier. With Seaborn, the `regplot()` function allows us to plot a trendline over the scatterplot, along with a 95% confidence interval for the trendline. Note that this is much easier than making a trendline with Matplotlib.

```
#Scatterplot and trendline with seaborn
plt.rcParams["figure.figsize"] = (9,6)
sns.set(font_scale=1.5)
ax=sns.regplot(x = 'gdpPercap', y = 'lifeExp', data = gdp_data,scatter_kws={"color": "oran
ax.xaxis.set_major_formatter('${x:,.0f}')
ax.set_xlabel('GDP per capita')
ax.set_ylabel('Life expectancy')
```

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



Note that the confidence interval of the trendline broadens as we move farther away from most of the data points. In other words, there is more uncertainty about the trend as we move to a domain space farther away from the data.

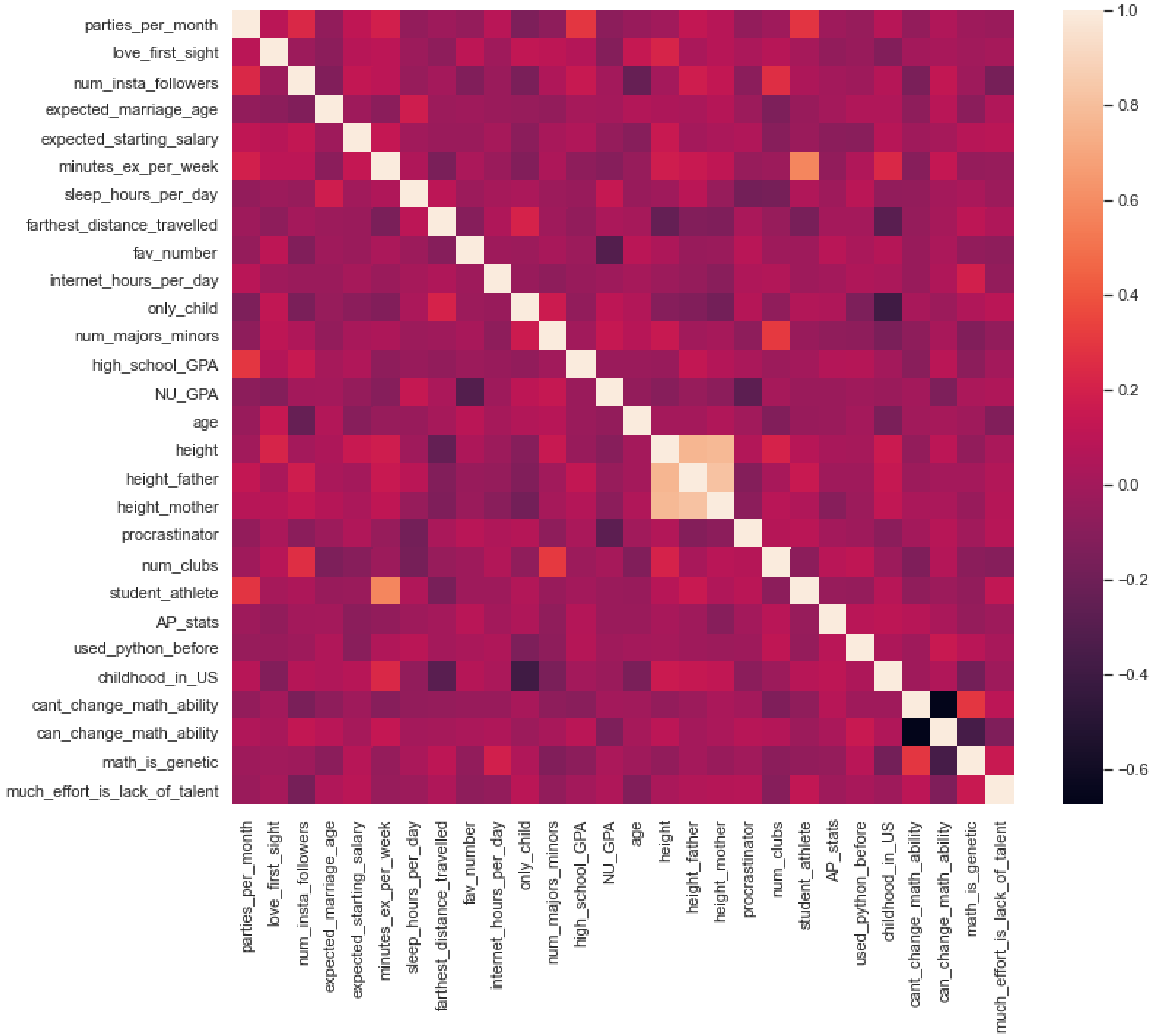
6.3.7 Heatmaps with Seaborn

Purpose: Heatmaps help us visualize the correlation between all variable-pairs.

Below is a heatmap visualizing the pairwise correlation of all the numerical variables of `survey_data_clean`. With a heatmap it becomes easier to see strongly correlated variables.

```
sns.set(rc={'figure.figsize':(12,10)})
sns.heatmap(survey_data.corr())
```

```
<AxesSubplot:>
```



From the above map, we can see that:

- `student athlete` is strongly postively correlated with `minutes_ex_per_week`

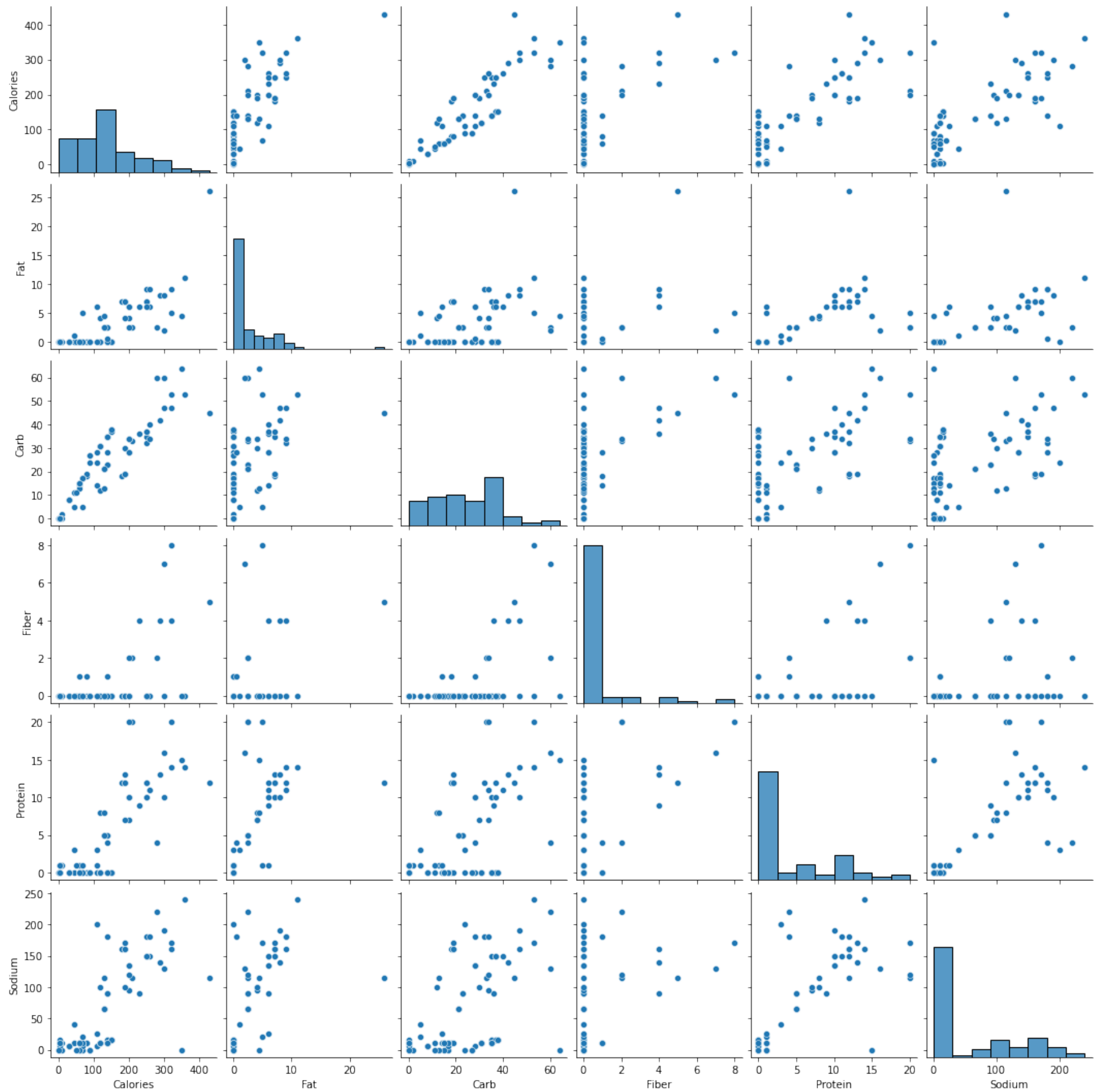
- `procrastinator` is strongly negatively correlated with `NU_GPA`

6.3.8 Pairplots with Seaborn

Purpose: Pairplots are used to visualize the association between all variable-pairs in the data. In other words, pairplots simultaneously visualize the scatterplots between all variable-pairs.

Let us visualize the pair-wise association of nutrition variables in the starbucks drinks data.

```
starbucks_drinks = pd.read_csv('./Datasets/starbucks-menu-nutrition-drinks.csv')
sns.pairplot(starbucks_drinks)
```



In the above pairplot, note that:

- The histograms on the diagonal of the grid show the distribution of each of the variables.
- Instead of a histogram, we can visualize the density plot with the argument *kde = True*.
- The scatterplots in the rest of the grid are the pair-wise plots of all the variables.

From the above plot, we observe that:

- Almost all the variable pairs have a positive correlation, i.e., if one of the nutrients increase in a drink, others also are likely to increase.
- The number of calories seem to be strongly positively correlated with the amount of carbs in the drink.
- From the density plots we can see that there is a lot of choice for consumers to buy a drink that has a zero value for any of the nutrients - fat, protein, fiber, or sodium.

7 Data cleaning and preparation

7.1 Handling missing data

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 researcher, etc.

Let us read the dataset *GDP_missing_data.csv*, in which we have randomly removed some values, or put missing values in some of the columns.

We'll also read *GDP_complete_data.csv*, in which we have not removed any values. We'll use this data later to assess the accuracy of our guess or estimate of missing values in *GDP_missing_data.csv*.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
import seaborn as sns
gdp_missing_values_data = pd.read_csv('./Datasets/GDP_missing_data.csv')
gdp_complete_data = pd.read_csv('./Datasets/GDP_complete_data.csv')
```

```
gdp_missing_values_data.head()
```

	economicActivityFemale	country	lifeMale	infantMortality	gdpPerCapita	economicActivityMale
0	7.2	Afghanistan	45.0	154.0	2474.0	87.5
1	7.8	Algeria	67.5	44.0	11433.0	76.4
2	41.3	Argentina	69.6	22.0	NaN	76.2
3	52.0	Armenia	67.2	25.0	13638.0	65.0
4	53.8	Australia	NaN	6.0	54891.0	NaN

Observe that the `gdp_missing_values_data` dataset consists of some missing values shown as NaN (Not a Number).

7.1.1 Identifying missing values (isnull())

Missing values in a Pandas DataFrame can be identified with the isnull() method. The Pandas Series object also consists of the isnull() method. For finding the number of missing values in each column of gdp_missing_values_data, we will sum up the missing values in each column of the dataset:

```
gdp_missing_values_data.isnull().sum()
```

```
economicActivityFemale    10
country                    0
lifeMale                  10
infantMortality           10
gdpPerCapita              10
economicActivityMale      10
illiteracyMale            10
illiteracyFemale          10
lifeFemale                10
geographic_location        0
contraception             71
continent                  0
dtype: int64
```

Note that the descriptive statistics methods associated with Pandas objects ignore missing values by default. Consider the summary statistics of gdp_missing_values_data:

```
gdp_missing_values_data.describe()
```

	economicActivityFemale	lifeMale	infantMortality	gdpPerCapita	economicActivityMale	illit
count	145.000000	145.000000	145.000000	145.000000	145.000000	145
mean	45.935172	65.491724	37.158621	24193.482759	76.563448	13.5
std	16.875922	9.099256	34.465699	22748.764444	7.854730	16.4
min	1.900000	36.000000	3.000000	772.000000	51.200000	0.00
25%	35.500000	62.900000	10.000000	6837.000000	72.000000	1.00
50%	47.600000	67.800000	24.000000	15184.000000	77.300000	6.60
75%	55.900000	72.400000	54.000000	35957.000000	81.600000	19.5
max	90.600000	77.400000	169.000000	122740.000000	93.000000	70.5

Observe that the count statistics report the number of non-missing values of each column in the data, as the number of rows in the data (see code below) is more than the number of non-missing values of all the variables in the above table. Similarly, for the rest of the statistics, such as `mean`, `std`, etc., the missing values are ignored.

```
#The dataset gdp_missing_values_data has 155 rows
gdp_missing_values_data.shape[0]
```

```
155
```

7.1.2 Types of missing values

Now that we know how to identify missing values in the dataset, let us learn about the types of missing values that can be there. Rubin (1976) classified missing values in three categories.

7.1.2.1 Missing Completely at Random (MCAR)

If the probability of being missing is the same for all cases, then the data are 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.

7.1.2.2 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

7.1.2.3 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

7.1.3 Practice exercise 1

7.1.3.1

In which of the above scenarios can we ignore the observations corresponding to missing values without the risk of skewing the analysis/trends in the data?

7.1.3.2

In which of the above scenarios will it be the more risky to impute or estimate missing values?

7.1.3.3

For the datset consisting of GDP per capita, think of hypothetical scenarios in which the missing values of GDP per capita can correspond to MCAR / MAR / MNAR.

7.1.4 Dropping observations with missing values (`dropna()`)

Sometimes our analysis requires that there should be no missing values in the dataset. For example, while building statistical models, we may require the values of all the predictor variables. The quickest way is to use the `dropna()` method, which drops the observations that even have a single missing value, and leaves only complete observations in the data.

Let us drop the rows containing even a single value from `gdp_missing_values_data`.

```
gdp_no_missing_data = gdp_missing_values_data.dropna()
```

```
#Shape of gdp_no_missing_data
gdp_no_missing_data.shape
```

```
(42, 12)
```

Dropping rows with even a single missing value has reduced the number of rows from 155 to 42! However, earlier we saw that all the columns except `contraception` had at most 10 missing values. Removing all rows / columns with even a single missing value results in loss of data that is non-missing in the respective rows/columns. Thus, it is typically a bad idea to drop observations with even a single missing value, except in cases where we have a very small number of missing-value observations.

If a few values of a column are missing, we can possibly estimate them using the rest of the data, so that we can (hopefully) maximize the information that can be extracted from the data. However, if most of the values of a column are missing, it may be harder to estimate its values.

In this case, we see that around 50% values of the `contraception` column is missing. Thus, we'll drop the column as it may be hard to impute its values based on a relatively small number of non-missing values.

```
#Deleting column with missing values in almost half of the observations
gdp_missing_values_data.drop(['contraception'],axis=1,inplace=True)
gdp_missing_values_data.shape
```

```
(155, 11)
```

7.1.5 Some methods to impute missing values

There are an unlimited number of ways to impute missing values. Some imputation methods are provided in the Pandas documentation.

The best way to impute them will depend on the problem, and the assumptions taken. Below are just a few examples.

7.1.5.1 Method 1: Naive Method

Filling the missing value of a column by copying the value of the previous non-missing observation.

```
#Filling missing values: Method 1- Naive way
gdp_imputed_data = gdp_missing_values_data.fillna(method = 'ffill')
```

```
#Checking if any missing values are remaining
gdp_imputed_data.isnull().sum()
```

```
economicActivityFemale    0
country                   0
lifeMale                  0
infantMortality           0
gdpPerCapita              0
economicActivityMale      0
```

```
illiteracyMale          1
illiteracyFemale        0
lifeFemale              0
geographic_location     0
continent               0
dtype: int64
```

After imputing missing values, note there is still one missing value for *illiteracyMale*. Can you guess why one missing value remained?

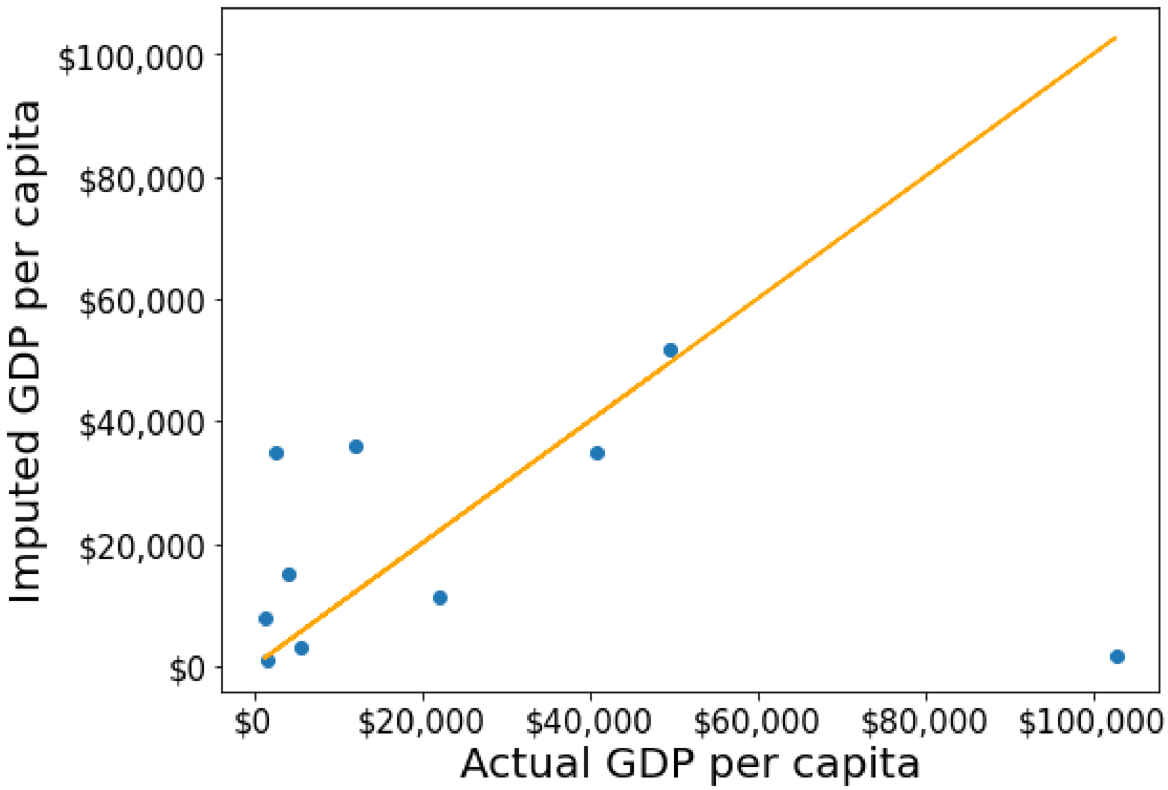
Let us check how good is this method in imputing missing values. We'll compare the imputed values of `gdpPerCapita` with the actual values. Recall that we had randomly put some missing values in `gdp_missing_values_data`, and we have the actual values in `gdp_complete_data`.

```
#Index of rows with missing values for GDP per capita
null_ind_gdpPC = gdp_missing_values_data.index[gdp_missing_values_data.gdpPerCapita.isnull
```

```
#Defining a function to plot the imputed values vs actual values
def plot_actual_vs_predicted():
    fig, ax = plt.subplots(figsize=(8, 6))
    plt.rc('xtick', labelsize=15)
    plt.rc('ytick', labelsize=15)
    x = gdp_complete_data.loc[null_ind_gdpPC,'gdpPerCapita']
    y = gdp_imputed_data.loc[null_ind_gdpPC,'gdpPerCapita']
    plt.scatter(x,y)
    z=np.polyfit(x,y,1)
    p=np.poly1d(z)
    plt.plot(x,x,color='orange')
    plt.xlabel('Actual GDP per capita',fontsize=20)
    plt.ylabel('Imputed GDP per capita',fontsize=20)
    ax.xaxis.set_major_formatter('${x:,.0f}')
    ax.yaxis.set_major_formatter('${x:,.0f}')
    rmse = np.sqrt(((x-y).pow(2)).mean())
    print("RMSE=",rmse)
```

```
#Plot comparing imputed values with actual values, and computing the Root mean square erro
plot_actual_vs_predicted()
```

```
RMSE= 34843.91091137732
```



We observe that the accuracy of imputation is poor as GDP per capita can vary a lot across countries, and the data is not sorted by GDP per capita. There is no reason why the GDP per capita of a country should be close to the GDP per capita of the country in the observation above it.

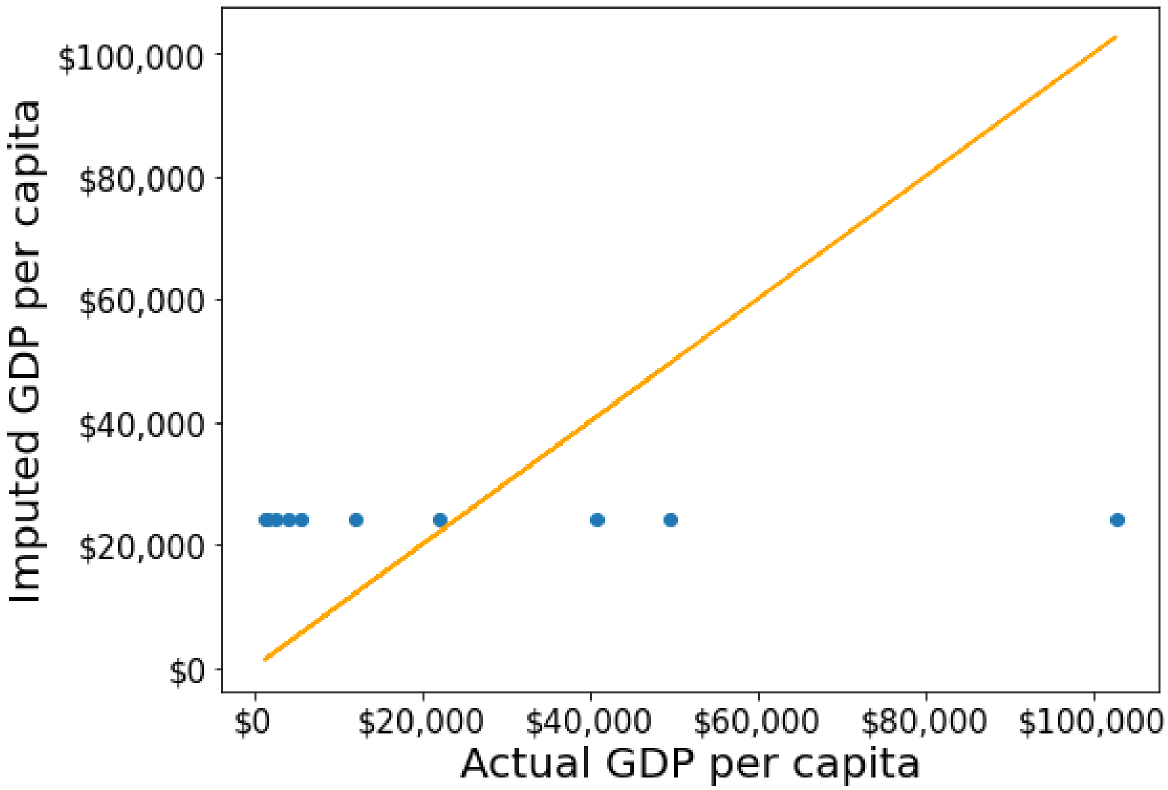
7.1.5.2 Method 2: Imputing missing values as the mean of non-missing values

Let us impute missing values in the column as the average of the non-missing values of the column. The sum of squared differences between actual values and the imputed values is likely to be smaller if we impute using the mean. However, this may not be true in cases other than MCAR (Missing completely at random).

```
#Filling missing values: Method 2
gdp_imputed_data = gdp_missing_values_data.fillna(gdp_missing_values_data.mean())
```

```
plot_actual_vs_predicted()
```

```
RMSE= 30793.549983587087
```



Although this method of imputation doesn't seem impressive, the RMSE of the estimates is lower than that of the naive method. Since we had introduced missing values randomly in `gdp_missing_values_data`, the mean GDP per capita will be the closest constant to the GDP per capita values, in terms of squared error.

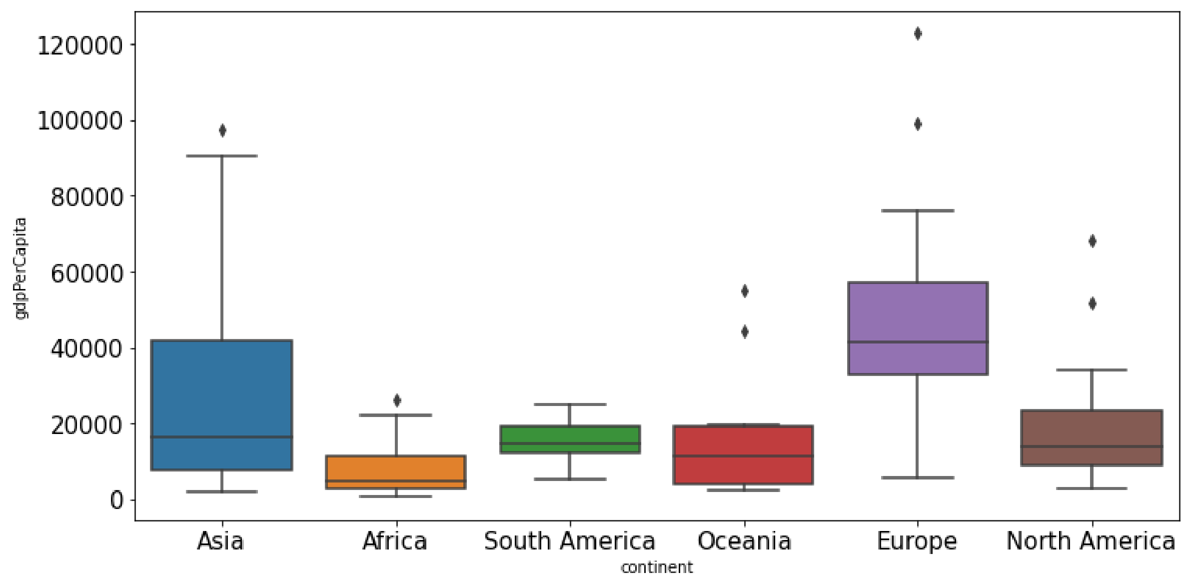
7.1.5.3 Method 3: Imputing missing values based on correlated variables in the data

If a variable is highly correlated with another variable in the dataset, we can approximate its missing values using the trendline with the highly correlated variable.

Let us visualize the distribution of GDP per capita for different continents.

```
plt.rcParams["figure.figsize"] = (12,6)
sns.boxplot(x = 'continent',y='gdpPerCapita',data = gdp_missing_values_data)
```

```
<AxesSubplot:xlabel='continent', ylabel='gdpPerCapita'>
```



We observe that there is a distinct difference between the GDPs per capita of some of the contents. Let us impute the missing GDP per capita of a country as the mean GDP per capita of the corresponding continent. This imputation should be better than imputing the missing GDP per capita as the mean of all the non-missing values, as the GDP per capita of a country is likely to be closer to the mean GDP per capita of the continent, rather the mean GDP per capita of the whole world.

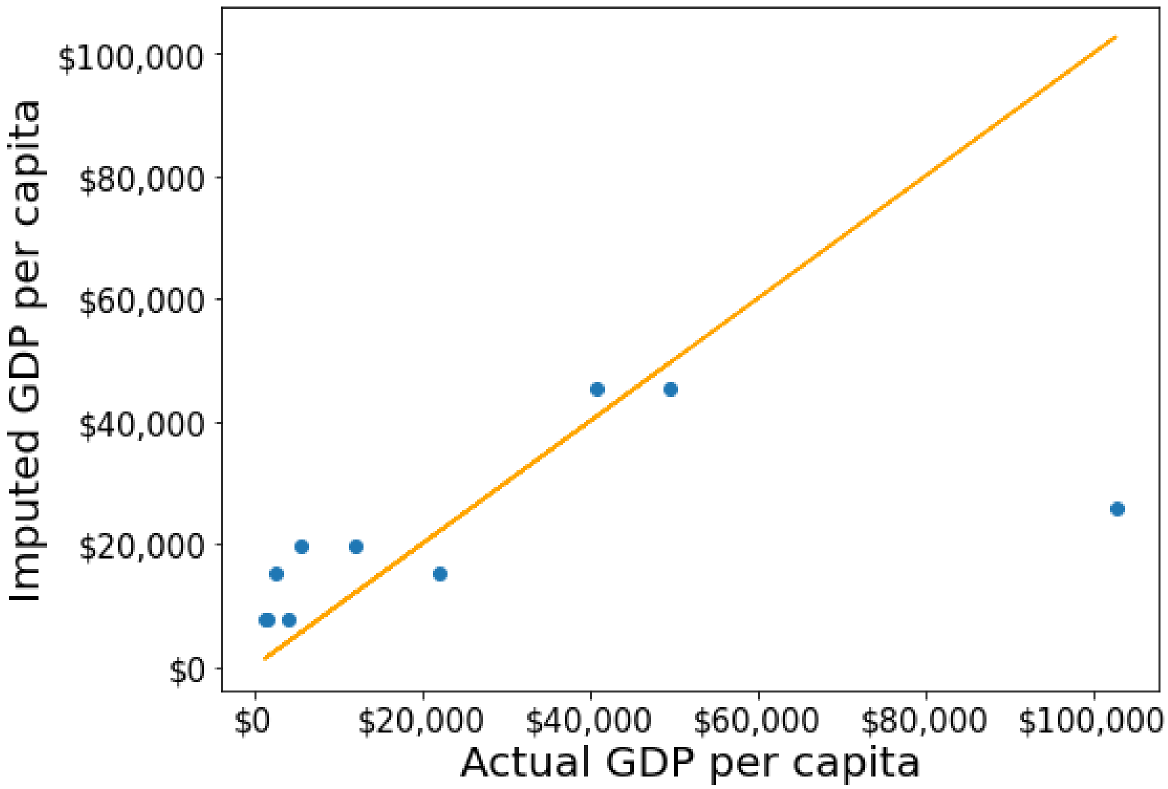
```
#Finding the mean GDP per capita of the continent - please defer the understanding of this
avg_gdpPerCapita = gdp_missing_values_data['gdpPerCapita'].groupby(gdp_missing_values_data
avg_gdpPerCapita
```

```
continent
Africa            7638.178571
Asia             25922.750000
Europe           45455.303030
North America    19625.210526
Oceania          15385.857143
South America    15360.909091
Name: gdpPerCapita, dtype: float64
```

```
#Creating a copy of missing data to impute missing values
gdp_imputed_data = gdp_missing_values_data.copy()
```

```
#Replacing missing GDP per capita with the mean GDP per capita for the corresponding conti
for cont in avg_gdpPerCapita.index:
    gdp_imputed_data.loc[(gdp_imputed_data.continent==cont) & (gdp_imputed_data.gdpPerCapi
                     'gdpPerCapita']=avg_gdpPerCapita[cont]
plot_actual_vs_predicted()
```

```
RMSE= 25473.20645170116
```



Note that the imputed values are closer to the actual values, and the RMSE has further reduced as expected.

7.1.5.4 Practice exercise 2

Find the numeric variable most strongly correlated with GDP per capita, and use it to impute its missing values. Find the RMSE of the imputed values.

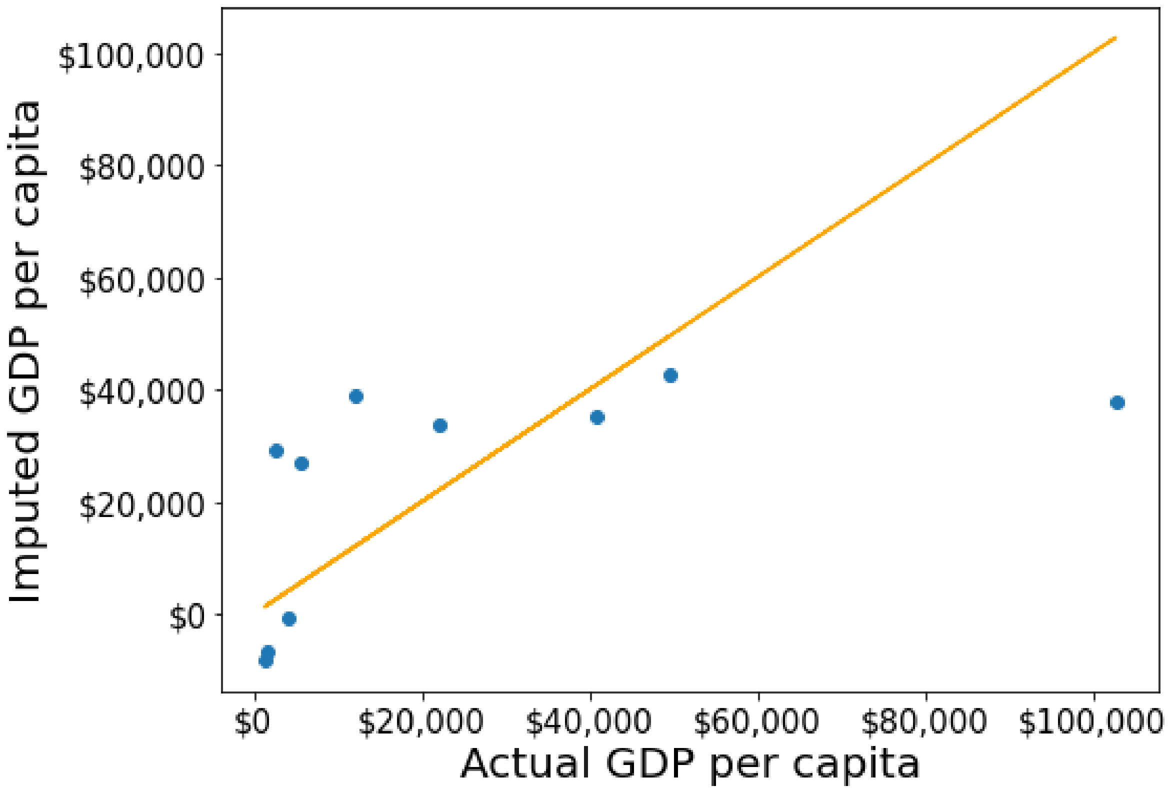
Solution:

```
#Let us identify the variable highly correlated with GDP per capita.
gdp_missing_values_data.corrwith(gdp_missing_values_data.gdpPerCapita)
```

```
economicActivityFemale    0.078332
lifeMale                  0.579850
infantMortality          -0.572201
gdpPerCapita              1.000000
economicActivityMale     -0.134108
illiteracyMale           -0.479143
illiteracyFemale         -0.448273
lifeFemale                0.615954
contraception             0.057923
dtype: float64
```

```
#The variable *lifeFemale* has the strongest correlation with GDP per capita. Let us use i
x = gdp_missing_values_data.lifeFemale
y = gdp_missing_values_data.gdpPerCapita
idx_non_missing = np.isfinite(x) & np.isfinite(y)
slope_intercept_trendline = np.polyfit(x[idx_non_missing],y[idx_non_missing],1)   #Finding
compute_y_given_x = np.poly1d(slope_intercept_trendline)
#Creating a copy of missing data to impute missing values
gdp_imputed_data = gdp_missing_values_data.copy()
gdp_imputed_data.loc[null_ind_gdpPC,'gdpPerCapita']=compute_y_given_x(gdp_missing_values_d
plot_actual_vs_predicted()
```

```
RMSE= 25570.361516956993
```



7.1.5.5 Method 4: KNN: K-nearest neighbor

In this method, we'll impute the missing value of the variable as the mean value of the K-nearest neighbors having non-missing values for that variable. The neighbors to a data-point are identified based on their Euclidean distance to the point in terms of the standardized values of rest of the variables in the data.

Let's consider a toy example to understand missing value imputation by KNN. Suppose we have to impute missing values in a toy dataset, named as `toy_data` having 4 observations and 3 variables.

```
#Toy example - A 4x3 array with missing values
nan = np.nan
toy_data = np.array([[1, 2, nan], [3, 4, 3], [nan, 6, 5], [8, 8, 7]])
toy_data
```

```
array([[ 1.,  2., nan],
       [ 3.,  4.,  3.],
```

```
       [nan,  6.,  5.],
       [ 8.,  8.,  7.]])
```

We'll use some functions from the *sklearn* library to perform the KNN imputation. It is much easier to directly use the algorithm from *sklearn*, instead of coding it from scratch.

```
#Library to compute pair-wise Euclidean distance between all observations in the data
from sklearn import metrics

#Library to impute missing values with the KNN algorithm
from sklearn import impute
```

We'll use the *sklearn* function nan_euclidean_distances() to compute the Euclidean distance between all pairs of observations in the data.

```
#This is the distance matrix containing the distance of the ith observation from the jth o
metrics.pairwise.nan_euclidean_distances(toy_data,toy_data)
```

```
array([[ 0.        ,  3.46410162,  6.92820323, 11.29158979],
       [ 3.46410162,  0.        ,  3.46410162,  7.54983444],
       [ 6.92820323,  3.46410162,  0.        ,  3.46410162],
       [11.29158979,  7.54983444,  3.46410162,  0.        ]])
```

Note that the size of the above matrix is 4x4. This is because the $(i,j)^{th}$ element of the matrix is the distance of the i^{th} observation from the j^{th} observation. The matrix is symmetric because the distance of i^{th} observation to the j^{th} observation is the same as the distance of the j^{th} observation to the i^{th} observation.

We'll use the *sklearn* function KNNImputer() to impute the missing value of a column in `toy_data` as the mean of the values of the K nearest neighbors to the observation that have non-missing values for that column.

Let us impute the missing values in `toy_data` using the values of $K = 2$ nearest neighbors from the corresponding observation.

```
#imputing missing values with 2 nearest neighbors, where the neighbors have equal weights

#Define an object of type KNNImputer
imputer = impute.KNNImputer(n_neighbors=2)

#Use the object method 'fit_transform' to impute missing values
imputer.fit_transform(toy_data)
```

```
array([[1. , 2. , 4. ],
       [3. , 4. , 3. ],
       [5.5, 6. , 5. ],
       [8. , 8. , 7. ]])
```

The third observation was the closest to the $2nd$ and $4th$ observations based on the Euclidean distance matrix. Thus, the missing value in the $3rd$ row of the `toy_data` has been imputed as the mean of the values in the $2nd$ and $4th$ observations for the corresponding column. Similarly, the $1st$ observation is the closest to the $2nd$ and $3rd$ observations. Thus the missing value in the $1st$ row of `toy_data` has been imputed as the mean of the values in the $1st$ and $2nd$ observations for the corresponding column.

Let us use KNN to impute the missing values of `gdpPerCapita` in `gdp_missing_values_data`. We'll use only the numeric columns of the data in imputing the missing values. Also, we'll ignore `contraception` as it has a lot of missing values, and thus may not be useful.

```
#Considering numeric columns in the data to use KNN
num_cols = list(range(0,1))+list(range(2,9))
num_cols
```

```
[0, 2, 3, 4, 5, 6, 7, 8]
```

Before computing the pair-wise Euclidean distance of observations, we must standardize the data so that all columns are at the same scale. This will avoid columns with a higher magnitude of values having a higher weight in determining the Euclidean distance. Unless there is a reason to give a higher weight to a column, we assume all columns to have the same weight in the Euclidean distance computation.

We can use the code below to scale the data. However, after imputing the missing values, the data is to be scaled back to the original scale, so that each variable is in the same units as in the original dataset. However, if the code below is used, we'll lose the orginal scale of each of the columns.

```
#Scaling data to compute equally weighted distances from the 'k' nearest neighbors
scaled_data = gdp_missing_values_data.iloc[:,num_cols].apply(lambda x:(x-x.min())/(x.max()
```

To alleviate the problem of losing the orignial scale of the data, we'll use the MinMaxScaler object of the *sklearn* library. The object will store the original scale of the data, which will help transform the data back to the original scale once the missing values have been imputed in the standardized data.

```python
# Scaling data - using sklearn

#Create an object of type MinMaxScaler
scaler = sk.preprocessing.MinMaxScaler()

#Use the object method 'fit_transform' to scale the values to a standard uniform distribut
scaled_data = pd.DataFrame(scaler.fit_transform(gdp_missing_values_data.iloc[:,num_cols]))
```

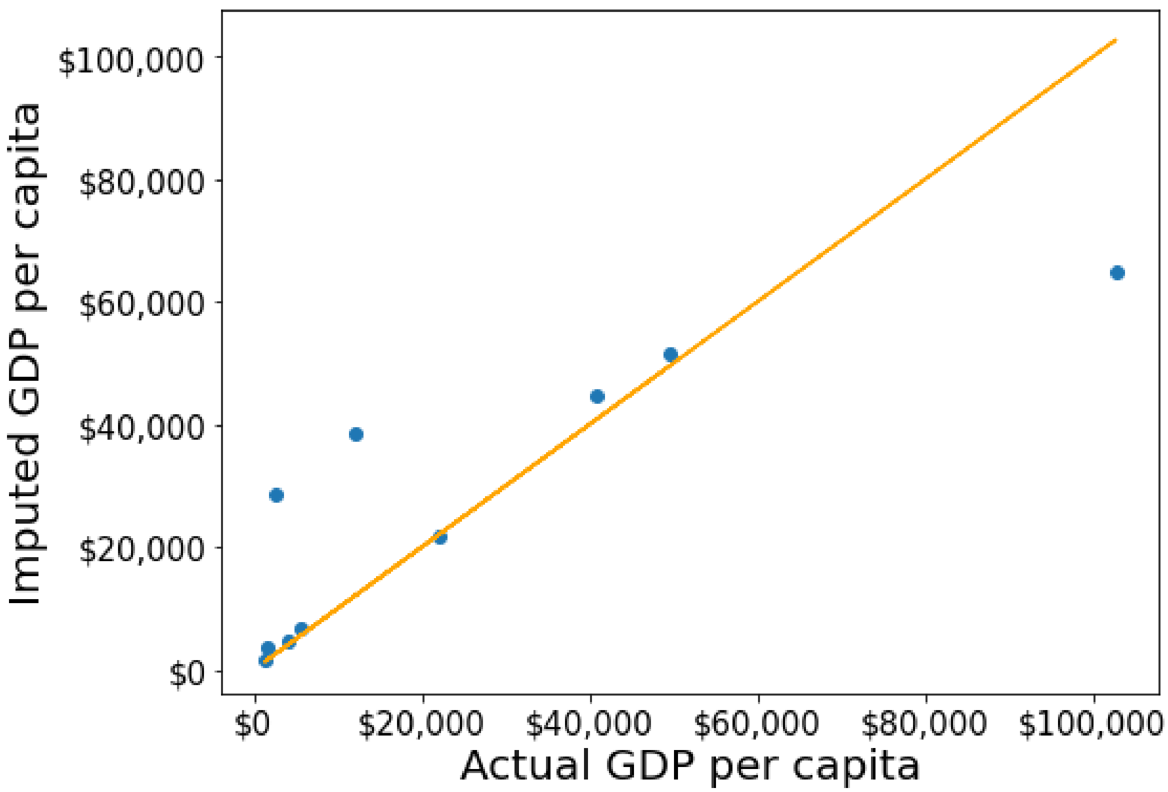
```python
#Imputing missing values with KNNImputer

#Define an object of type KNNImputer
imputer = impute.KNNImputer(n_neighbors=3, weights="uniform")

#Use the object method 'fit_transform' to impute missing values
imputed_arr = imputer.fit_transform(scaled_data)
```

```python
#Scaling back the scaled array to obtain the data at the original scale

#Use the object method 'inverse_transform' to scale back the values to the original scale
unscaled_data = scaler.inverse_transform(imputed_arr)
```

```python
#Note the method imputes the missing value of all the columns
#However, we are interested in imputing the missing values of only the 'gdpPerCapita' colu
gdp_imputed_data = gdp_missing_values_data.copy()
gdp_imputed_data.loc[:,'gdpPerCapita'] = unscaled_data[:,3]
```

```python
#Visualizing the accuracy of missing value imputation with KNN
plot_actual_vs_predicted()
```

```
RMSE= 16804.195967740387
```



Note that the RMSE is the lowest in this method. It is because this method imputes missing values as the average of the values of "similar" observations, which is smarter and more robust than the previous methods.

We chose $K = 3$ in the missing value imputation for GDP per capita. However, the value of K is typically chosen using a method known as cross validation. We'll learn about cross-validation in the next course of the sequence.

7.2 Data binning

Data binning is a method to group values of a continuous / categorical variable into bins (or categories). Binning may help with

(i) Better intepretation of data

(ii) Making better recommendations

(iii) Smooth data, reduce noise

Examples:

Binning to better interpret data

1. The number of flu cases everyday may be binned to seasons such as fall, spring, winter and summer, to understand the effect of season on flu.

Binning to make recommendations:

2. A doctor may like to group patient age into bins. Grouping patient ages into categories such as Age <=12, 12<Age<=18, 18<Age<=65, Age>65 may help recommend the kind/doses of covid vaccine a patient needs.

3. A credit card company may want to bin customers based on their spend, as "High spenders", "Medium spenders" and "Low spenders". Binning will help them design customized marketing campaigns for each bin, thereby increasing customer response (or revenue). On the other hand, they use the same campaign for customers withing the same bin, thus minimizng marketing costs.

Binning to smooth data, and reduce noise

4. A sales company may want to bin their total sales to a weekly / monthly / yearly level to reduce the noise in day-to-day sales.

Example: The dataset *College.csv* contains information about US universities. The description of variables of the dataset can be found on page 54 of this book. Let's see if we can apply binning to better interpret the association of instructional expenditure per student (`Expend`) with graduation rate (`Grad.Rate`) for US universities, and make recommendations.

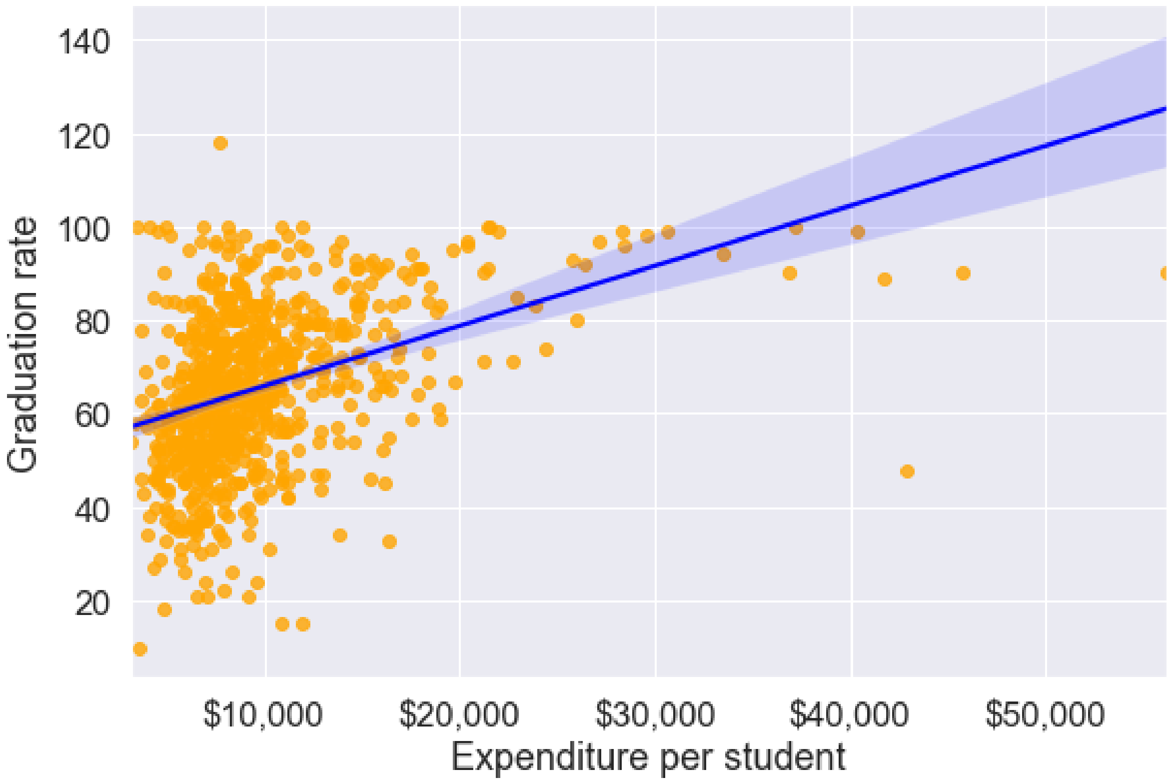
```
college = pd.read_csv('./Datasets/College.csv')
college.head()
```

	Unnamed: 0	Private	Apps	Accept	Enroll	Top10perc	Top25perc	F.Undergrad
0	Abilene Christian University	Yes	1660	1232	721	23	52	2885
1	Adelphi University	Yes	2186	1924	512	16	29	2683
2	Adrian College	Yes	1428	1097	336	22	50	1036
3	Agnes Scott College	Yes	417	349	137	60	89	510
4	Alaska Pacific University	Yes	193	146	55	16	44	249

To visualize the association between two numeric variables, we typically make a scatterplot. Let us make a scatterplot of graduation rate with expenditure per student, with a trendline.

```
#Let's make a scatterplot of 'Grad.Rate' vs 'Expend' with a trendline, to visualize any tr
sns.set(font_scale=1.5)
ax=sns.regplot(data = college, x = "Expend", y = "Grad.Rate",scatter_kws={"color": "orange
ax.xaxis.set_major_formatter('${x:,.0f}')
ax.set_xlabel('Expenditure per student')
ax.set_ylabel('Graduation rate')
```

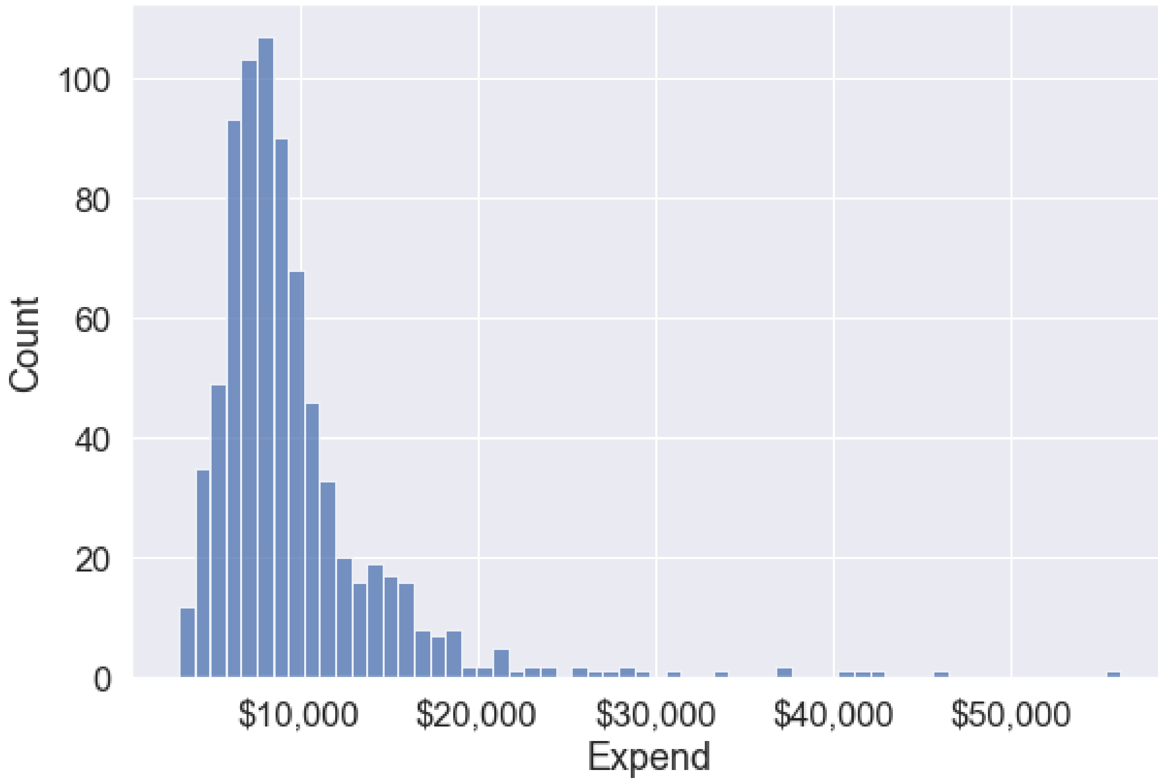
```
Text(0, 0.5, 'Graduation rate')
```



The trendline indicates a positive correlation between `Expend` and `Grad.Rate`. However, there seems to be a lot of noise and presence of outliers in the data, which makes it hard to interpret the overall trend.

We'll bin `Expend` to see if we can better analyze its association with `Grad.Rate`. However, let us first visualize the distribution of `Expend`.

```
#Visualizing the distribution of expend
ax=sns.histplot(data = college, x= 'Expend')
ax.xaxis.set_major_formatter('${x:,.0f}')
```



The distribution of `Extend` is right skewed with potentially some extremely high outlying values.

7.2.1 Binning with equal width bins

We'll use the Pandas function cut() to bin `Expend`. This function creates bins such that all bins have the same width.

```
#Using the cut() function in Pandas to bin "Expend"
Binned_expend = pd.cut(college['Expend'],3,retbins = True)
Binned_expend
```

```
(0      (3132.953, 20868.333]
```

```
1      (3132.953, 20868.333]
2      (3132.953, 20868.333]
3      (3132.953, 20868.333]
4      (3132.953, 20868.333]
                ...
772    (3132.953, 20868.333]
773    (3132.953, 20868.333]
774    (3132.953, 20868.333]
775     (38550.667, 56233.0]
776    (3132.953, 20868.333]
Name: Expend, Length: 777, dtype: category
Categories (3, interval[float64]): [(3132.953, 20868.333] < (20868.333, 38550.667] < (38550
array([ 3132.953     , 20868.33333333, 38550.66666667, 56233.        ]))
```

The cut() function returns a tuple of length 2. The first element of the tuple are the bins, while the second element is an array containing the cut-off values for the bins.

```
type(Binned_expend)
```

```
tuple
```

```
len(Binned_expend)
```

```
2
```

Once the bins are obtained, we'll add a column in the dataset that indicates the bin for `Expend`.

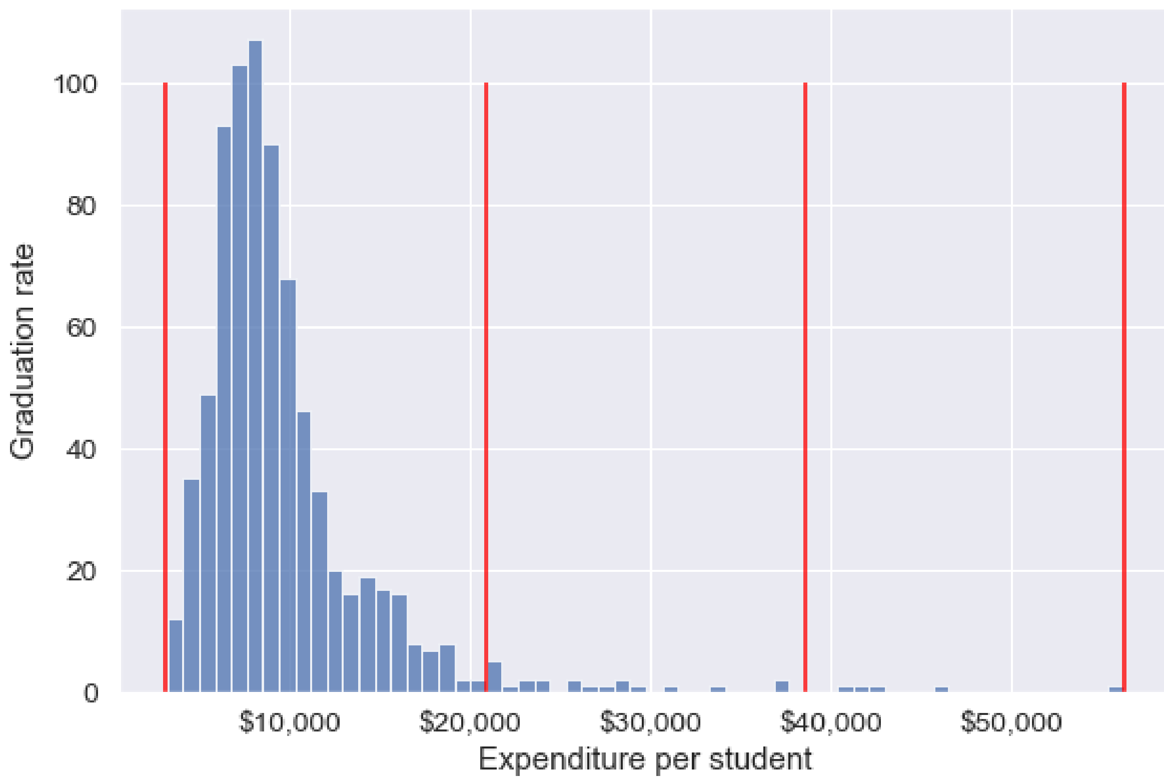
```
#Creating a categorical variable to store the level of expenditure on a student
college['Expend_bin'] = Binned_expend[0]
college.head()
```

	Unnamed: 0	Private	Apps	Accept	Enroll	Top10perc	Top25perc	F.Undergrad
0	Abilene Christian University	Yes	1660	1232	721	23	52	2885
1	Adelphi University	Yes	2186	1924	512	16	29	2683
2	Adrian College	Yes	1428	1097	336	22	50	1036
3	Agnes Scott College	Yes	417	349	137	60	89	510
4	Alaska Pacific University	Yes	193	146	55	16	44	249

See the variable `Expend_bin` in the above dataset.

Let us visualize the `Expend` bins over the distribution of the `Expend` variable.

```
#Visualizing the bins for instructional expediture on a student
sns.set(font_scale=1.25)
plt.rcParams["figure.figsize"] = (9,6)
ax=sns.histplot(data = college, x= 'Expend')
plt.vlines(Binned_expend[1], 0,100,color='red')
plt.xlabel('Expenditure per student');
plt.ylabel('Graduation rate');
ax.xaxis.set_major_formatter('${x:,.0f}')
```



By default, the bins created have equal width. They are created by dividing the range between the maximum and minimum value of `Expend` into the desired number of equal-width intervals. We can label the bins as well as follows.

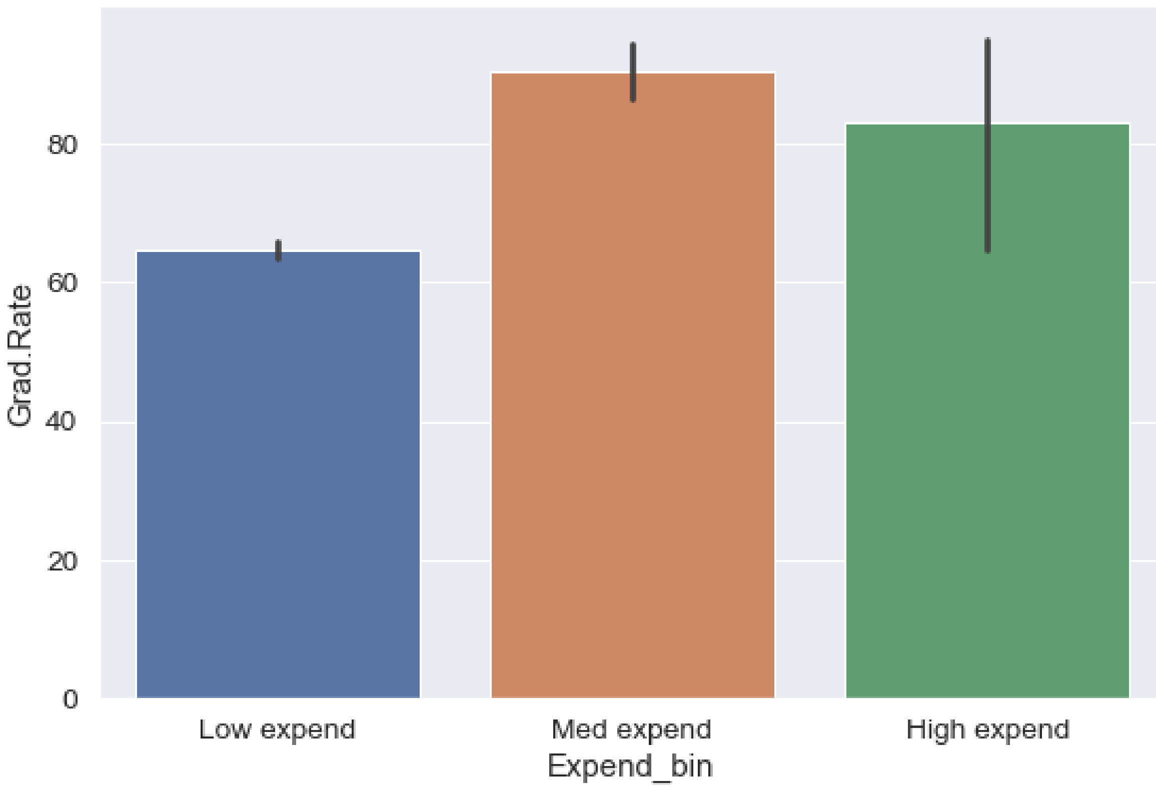
```
college['Expend_bin'] = pd.cut(college['Expend'],3,labels = ['Low expend','Med expend','Hi
college['Expend_bin']
```

```
0        Low expend
1        Low expend
2        Low expend
3        Low expend
4        Low expend
           ...
772      Low expend
773      Low expend
774      Low expend
775     High expend
776      Low expend
Name: Expend_bin, Length: 777, dtype: category
Categories (3, object): ['Low expend' < 'Med expend' < 'High expend']
```

Now that we have binned the variable `Expend`, let us see if we can better visualize the association of graduation rate with expenditure per student using `Expened_bin`.

```
#Visualizing average graduation rate vs categories of instructional expenditure per studen
sns.barplot(x = 'Expend_bin', y = 'Grad.Rate', data = college)
```

```
<AxesSubplot:xlabel='Expend_bin', ylabel='Grad.Rate'>
```



It seems that the graduation rate is the highest for universities with medium level of expenditure per student. This is different from the trend we saw earlier in the scatter plot. Let us investigate.

Let us find the number of universities in each bin.

```
pd.value_counts(college['Expend_bin'])
```

```
Low expend     751
Med expend      21
High expend      5
Name: Expend_bin, dtype: int64
```

The bin `High expend` consists of only 5 universities, or 0.6% of all the universities in the dataset. These universities may be outliers that are skewing the trend (as also evident in the histogram above).

In such cases, we should bin observations such that all bins are of equal size, i.e., they have the same number of observations.

7.2.2 Binning with equal sized bins

Let us bin the variable `Expend` such that each bin consists of the same number of observations.

We'll use the Pandas function qcut() to make equal-sized bins (in contrast to equal-width bins in the previous section).

```
#Using the Pandas function qcut() to create bins with the same number of observations
Binned_expend = pd.qcut(college['Expend'],3,retbins = True)
college['Expend_bin'] = Binned_expend[0]
```

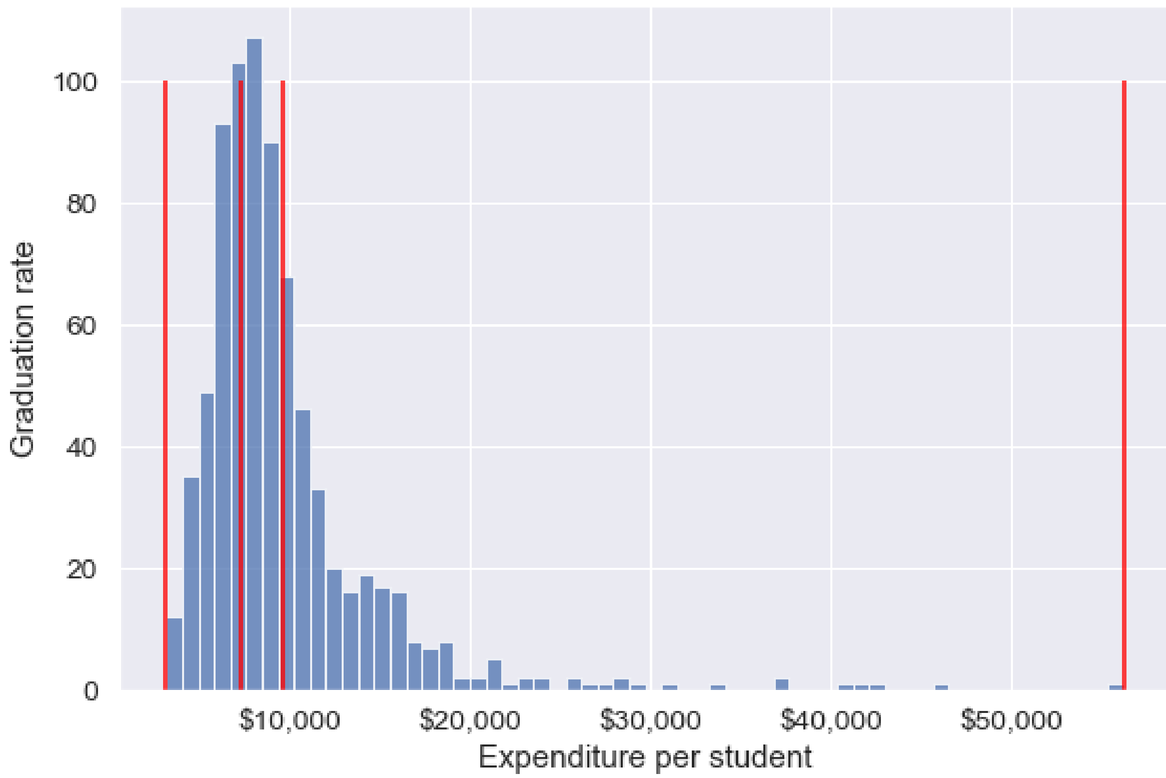
Each bin has the same number of observations with `qcut()`:

```
pd.value_counts(college['Expend_bin'])
```

```
(3185.999, 7334.333]    259
(7334.333, 9682.333]    259
(9682.333, 56233.0]     259
Name: Expend_bin, dtype: int64
```

Let us visualize the `Expend` bins over the distribution of the `Expend` variable.

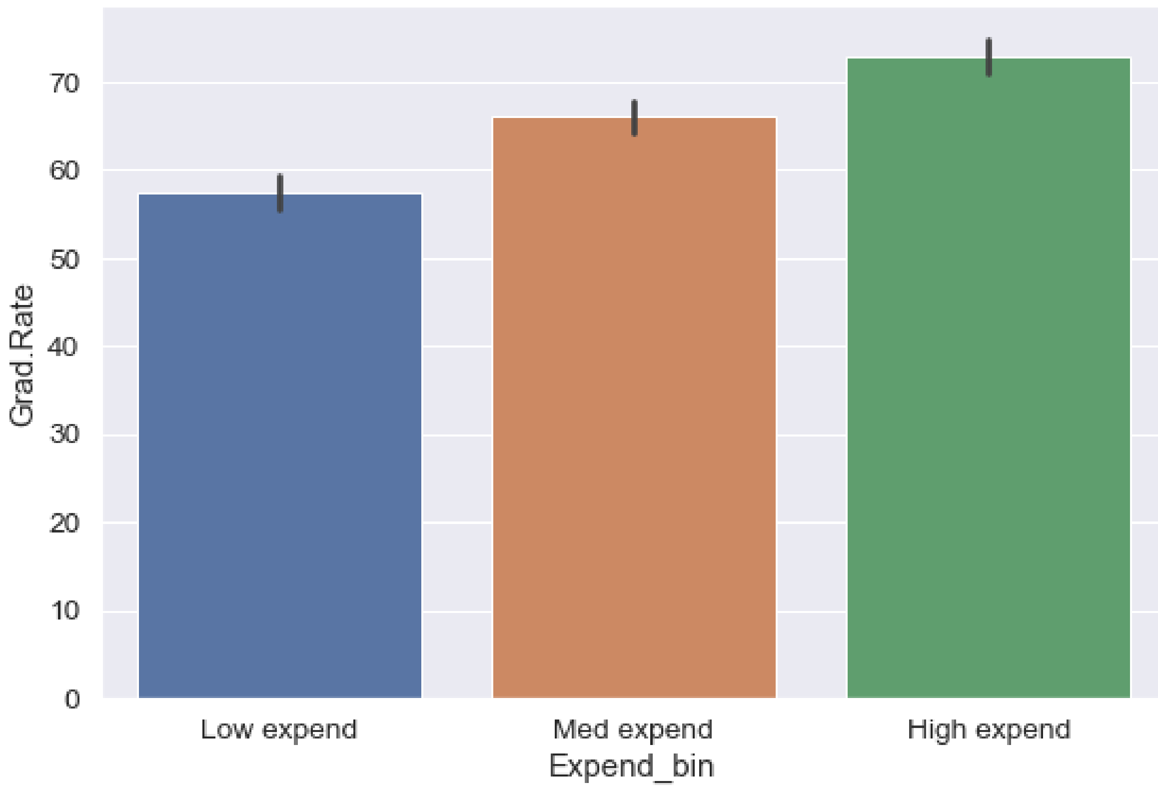
```
#Visualizing the bins for instructional expediture on a student
sns.set(font_scale=1.25)
plt.rcParams["figure.figsize"] = (9,6)
ax=sns.histplot(data = college, x= 'Expend')
plt.vlines(Binned_expend[1], 0,100,color='red')
plt.xlabel('Expenditure per student');
plt.ylabel('Graduation rate');
ax.xaxis.set_major_formatter('${x:,.0f}')
```



Note that the bin-widths have been adjusted to have the same number of observations in each bin. The bins are narrower in domains of high density, and wider in domains of sparse density.

Let us again make the barplot visualizing the average graduate rate with level of instructional expenditure per student.

```
college['Expend_bin'] = pd.qcut(college['Expend'],3,labels = ['Low expend','Med expend','H
a=sns.barplot(x = 'Expend_bin', y = 'Grad.Rate', data = college)
```



Now we see the same trend that we saw in the scatterplot, but without the noise. We have smoothed the data. Note that making equal-sized bins helps reduce the effect of outliers in the overall trend.

Suppose this analysis was done to provide recommendations to universities for increasing their graduation rate. With binning, we can can provide one recommendation to *'Low expend'* universities, and another one to *'Med expend'* universities. For example, the recommendations can be:

1. *'Low expend'* universities can expect an increase of 9 percentage points in `Grad.Rate`, if they migrate to the *'Med expend'* category.
2. *'Med expend'* universities can expect an increase of 7 percentage points in `Grad.Rate`, if they migrate to the *'High expend'* category.

The numbers in the above recommendations are based on the table below.

```
college['Grad.Rate'].groupby(college.Expend_bin).mean()
```

```
Expend_bin
```

```
Low expend      57.343629
Med expend      66.057915
High expend     72.988417
Name: Grad.Rate, dtype: float64
```

We can also make recommendations based on the confidence intervals of mean Grad.Rate. Confidence intervals are computed below. We are finding confidence intervals based on a method known as bootstrapping. Refer https://en.wikipedia.org/wiki/Bootstrapping_(statistics) for a detailed description of Bootstrapping.

```
#Bootstrapping to find 95% confidence intervals of Graduation Rate of US universities base
for expend_bin in college.Expend_bin.unique():
    data_sub = college.loc[college.Expend_bin==expend_bin,:]
    samples = np.random.choice(data_sub['Grad.Rate'], size=(10000,data_sub.shape[0]))
    print("95% Confidence interval of Grad.Rate for "+expend_bin+" univeristies = ["+str(n
```

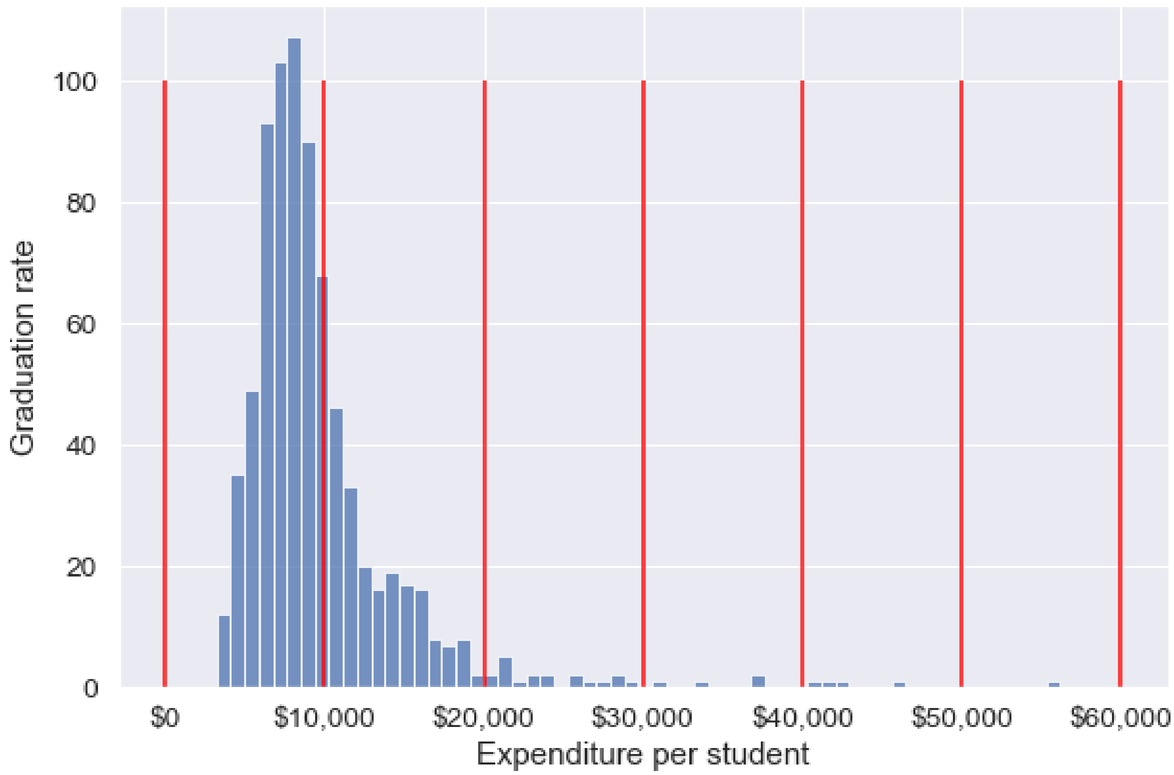
```
95% Confidence interval of Grad.Rate for Low expend univeristies = [55.34,59.34]
95% Confidence interval of Grad.Rate for High expend univeristies = [71.01,74.92]
95% Confidence interval of Grad.Rate for Med expend univeristies = [64.22,67.93]
```

Apart from equal-width and equal-sized bins, custom bins can be created using the *bins* argument. Suppose, bins are to be created for `Expend` with cutoffs $10, 000, $20, 000, $30, 000...$60, 000. Then, we can use the *bins* argument as in the code below:

7.2.3 Binning with custom bins

```
Binned_expend = pd.cut(college.Expend,bins = list(range(0,70000,10000)),retbins=True)
```

```
#Visualizing the bins for instructional expediture on a student
ax=sns.histplot(data = college, x= 'Expend')
plt.vlines(Binned_expend[1], 0,100,color='red')
plt.xlabel('Expenditure per student');
plt.ylabel('Graduation rate');
ax.xaxis.set_major_formatter('${x:,.0f}')
```



As custom bin-cutoffs can be specified with the `cut()` function, custom bin quantiles can be specified with the `qcut()` function.

7.3 Dummy / Indicator variables

Dummy variables (or indicator variables) take only the values of 0 and 1 to indicate the presence or absence of a catagorical effect. They are particularly useful in regression modeling to help explain the dependent variable.

If a column in a DataFrame has k distinct values, we will get a DataFrame with k columns containing 0s and 1s with the Pandas `get_dummies()` function.

Let us make dummy variables with the equal-sized bins we created for the average instruction expenditure per student.

```
#Using the Pandas function qcut() to create bins with the same number of observations
Binned_expend = pd.qcut(college['Expend'],3,retbins = True,labels = ['Low_expend','Med_exp
college['Expend_bin'] = Binned_expend[0]
```

```
#Making dummy variables based on the levels (categories) of the 'Expend_bin' variable
dummy_Expend = pd.get_dummies(college['Expend_bin'])
```

The dummy data `dummy_Expend` has a value of 1 if the observation corresponds to the category referenced by the column name.

```
dummy_Expend.head()
```

	Low_expend	Med_expend	High_expend
0	1	0	0
1	0	0	1
2	0	1	0
3	0	0	1
4	0	0	1

We can find the correlation between the dummy variables and graduation rate to identify if any of the dummy variables will be useful to estimate graduation rate (`Grad.Rate`).

```
#Finding if dummy variables will be useful to estimate 'Grad.Rate'
dummy_Expend.corrwith(college['Grad.Rate'])
```

```
Low_expend    -0.334456
Med_expend     0.024492
High_expend    0.309964
dtype: float64
```

The dummy variables `Low expend` and `High expend` may contribute in explaining `Grad.Rate` in a regression model.

7.3.1 Practice exercise 3

Read *survey_data_clean.csv*. Split the columns of the dataset, such that all columns with categorical values transform into dummy variables with each category corresponding to a column of 0s and 1s. Leave the *Timestamp* column.

As all categorical columns are transformed to dummy variables, all columns have numeric values.

What is the total number of columns in the transformed data? What is the total number of columns of the original data?

Find the:

1. Top 5 variables having the highest positive correlation with `NU_GPA`.
2. Top 5 variables having the highest negative correlation with `NU_GPA`.

Solution:

```
survey_data = pd.read_csv('./Datasets/survey_data_clean.csv')
survey_dummies=pd.get_dummies(survey_data.iloc[:,1:])
print("The total number of columns in the transformed data are", survey_dummies.shape[1])
print("The total number of columns in the original data are", survey_dummies.shape[0])
```

```
The total number of columns in the transformed data are 308
The total number of columns in the original data are 192
```

Below are the top 5 variables having the highest positive correlation with `NU_GPA`:

```
survey_dummies.corrwith(survey_dummies.NU_GPA).drop(index = 'NU_GPA').sort_values(ascendin
```

```
fav_letter_o                                                                        0.3
fav_sport_Dance!                                                                    0.3
major_Humanities / Communications;Physical Sciences / Natural Sciences / Engineering     0.2
fav_alcohol_I don't drink                                                           0.2
learning_style_Reading/Writing (learn best through words often note-taking or reading)    0.2
dtype: float64
```

Below are the top 5 variables having the highest negative correlation with `NU_GPA`:

```
survey_dummies.corrwith(survey_dummies.NU_GPA).drop(index = 'NU_GPA').sort_values(ascendin
```

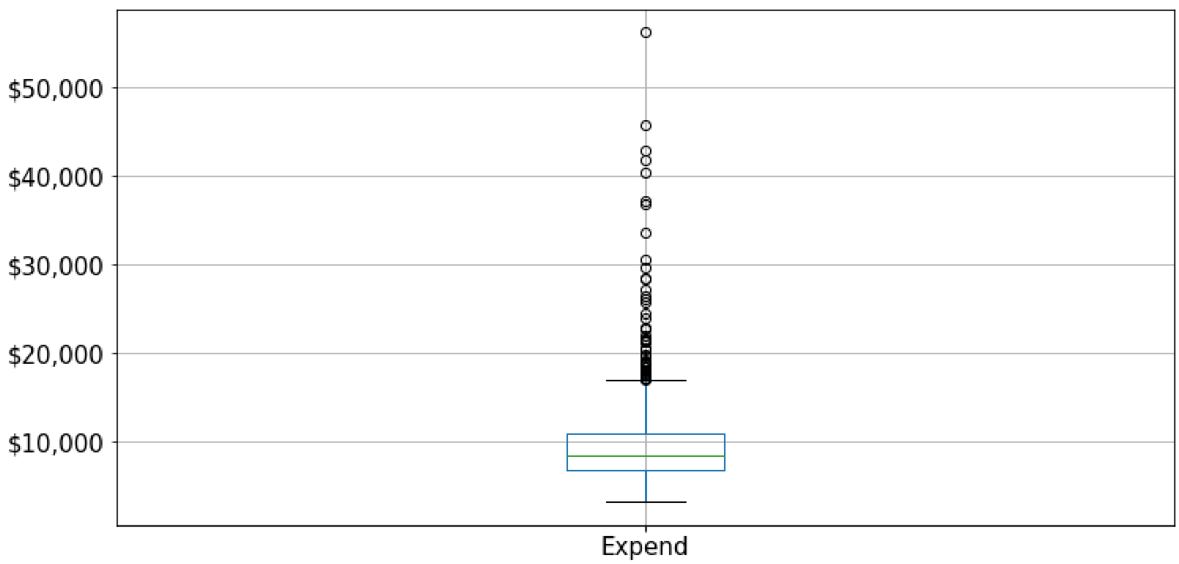
```
fav_number                                    -0.307656
procrastinator                                -0.269552
fav_sport_Underwater Basketweaving            -0.224237
birth_month_February                          -0.222141
streaming_platforms_Netflix;Amazon Prime;HBO Max   -0.221099
dtype: float64
```

7.4 Outlier detection

An outlier is an observation that is significantly different from the rest of the data. Detection of outliers is important as they may distort the general trends in data.

Let us visualize outliers in average instructional expenditure per student given by the variable `Expend`.

```
ax=college.boxplot(column = 'Expend')
ax.yaxis.set_major_formatter('${x:,.0f}')
```



There are several outliers (shown as circles in the above boxplot), which correspond to high values of average instructional expenditure per student. Boxplot identifies outliers based on the Tukey's fences criterion:

Tukey's fences: John Tukey proposed that observations outside the range $[Q1 - 1.5(Q3 - Q1), Q3 + 1.5(Q3 - Q1)]$ are outliers, where $Q1$ and $Q3$ are the lower (25%) and upper (75%) quartiles respectively. Let us detect outliers based on Tukey's fences.

```
#Finding upper and lower quartiles and interquartile range
q1 = np.percentile(college['Expend'],25)
q3 = np.percentile(college['Expend'],75)
intQ_range = q3-q1
```

```
#Tukey's fences
Lower_fence = q1 - 1.5*intQ_range
Upper_fence = q3 + 1.5*intQ_range
```

```
#These are the outlying observations - those outside of Tukey's fences
Outlying_obs = college[(college.Expend<Lower_fence) | (college.Expend>Upper_fence)]
```

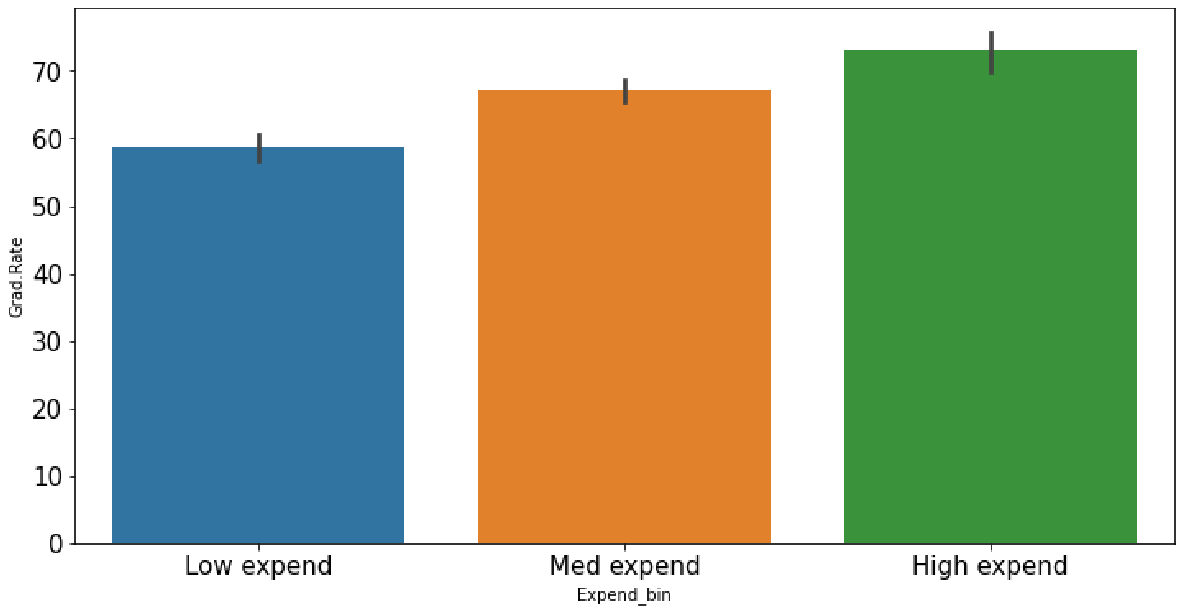
```
#Data without outliers
college_data_without_outliers = college[((college.Expend>=Lower_fence) & (college.Expend<=
```

Earlier, the trend was distorted by outliers when we created bins of equal width. Let us see if we get the correct trend with the outliers removed from the data.

```
Binned_data = pd.cut(college_data_without_outliers['Expend'],3,labels = ['Low expend','Med
college_data_without_outliers.loc[:,'Expend_bin'] = Binned_data[0]
```

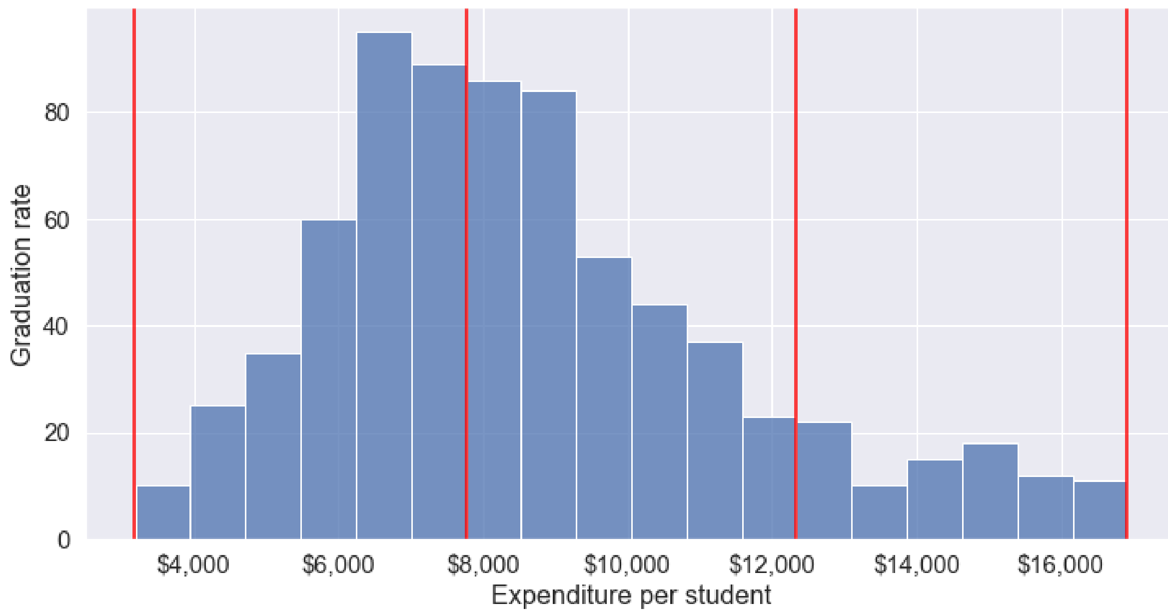
```
sns.barplot(x = 'Expend_bin', y = 'Grad.Rate', data = college_data_without_outliers)
```

```
<AxesSubplot:xlabel='Expend_bin', ylabel='Grad.Rate'>
```



With the outliers removed, we obtain the correct overall trend, even in the case of equal-width bins. Note that these bins have unequal number of observations as shown below.

```
sns.set(font_scale=1.35)
ax=sns.histplot(data = college_data_without_outliers, x= 'Expend')
for i in range(4):
    plt.axvline(Binned_data[1][i], 0,100,color='red')
plt.xlabel('Expenditure per student');
plt.ylabel('Graduation rate');
ax.xaxis.set_major_formatter('${x:,.0f}')
```



Note that the right tail of the histogram has disappered since we removed outliers.

```
college_data_without_outliers['Expend_bin'].value_counts()
```

```
Med expend     327
Low expend     314
High expend     88
Name: Expend_bin, dtype: int64
```

7.4.1 Practice exercise 4

Consider the dataset created for *survey_data_clean.csv* in Practice exercise 3, which includes dummy variables for all the categorical variables. Find the number of outliers in each column of the dataset based on the Tukey's fences criterion. Do not use a `for` loop.

Which column(s) have the maximum number of outliers?

Do you think the outlying observations identified with the Tukey's fences criterion for those columns(s) should be considered as outliers? If not, then which type of columns should be considered when finding outliers?

Solution:

```
#Function to identify outliers based on Tukey's fences
def rem_outliers(x):
    q1 =x.quantile(0.25)
    q3 = x.quantile(0.75)
    intQ_range = q3-q1

    #Tukey's fences
    Lower_fence = q1 - 1.5*intQ_range
    Upper_fence = q3 + 1.5*intQ_range

    #The object returned will be a data frame with bool values - True or False. 'True' wil
    return ((x<Lower_fence) | (x>Upper_fence))

survey_dummies.apply(rem_outliers).sum().sort_values(ascending=False)
```

```
learning_style_Reading/Writing (learn best through words often note-taking or reading)    48
much_effort_is_lack_of_talent                                                             48
left_right_brained_Left-brained (logic, science, critical thinking, numbers)              47
left_right_brained_Right-brained (creative, art, imaginative, intuitive)                  47
fav_season_Spring                                                                         47
                                                                                          ..
love_first_sight                                                                           0
birthdate_odd_even_Odd                                                                     0
procrastinator                                                                             0
birthdate_odd_even_Even                                                                    0
how_happy_Pretty happy                                                                     0
Length: 308, dtype: int64
```

Using Tukey's criterion, the variables `learning_style_Reading/Writing (learn best through words often note-taking or reading)` and `much_effort_is_lack_of_talent` have the most number of outliers.

However, these two variables only have 0s and 1s. For instance, let us consider `learning_style_Reading/Writing (learn best through words often note-taking or reading)`.

```
survey_dummies['learning_style_Reading/Writing (learn best through words often note-taking
```

```
0    144
1     48
Name: learning_style_Reading/Writing (learn best through words often note-taking or reading)
```

As the percentage of 1s are $\frac{48}{48+144} = 25\%$, the 75^{th} percentile value is 0.25, and the upper Tukey's fence is $0.25 + 0.25 * 1.25 = 0.625$, which makes the value 1 an outlier. However, we should not consider this value as an outlier, as a considerable fraction of the data (25%) has the value 1 for this variable.

Furthermore, Tukey's fences are developed for continuous variables. However, the variable `learning_style_Reading/Writing (learn best through words often note-taking or reading)` is discrete with only two levels. Thus, while finding outliers we must consider only continuous variables.

```
#Finding continuous variables: Assuming numeric variables that have more than 2 distinct v
continuous_variables = [x for x in list(survey_data.apply(lambda x: x.name if ((len(x.valu
```

```
#Finding number of outliers for only continuous variables
survey_data.loc[:,continuous_variables].apply(rem_outliers).sum().sort_values(ascending=Fa
```

```
num_clubs                     22
fav_number                    19
parties_per_month             12
internet_hours_per_day        11
sleep_hours_per_day           10
expected_marriage_age         10
expected_starting_salary       8
high_school_GPA                8
height_mother                  7
num_insta_followers            6
height_father                  6
NU_GPA                         5
minutes_ex_per_week            4
height                         3
farthest_distance_travelled    2
age                            1
num_majors_minors              0
dtype: int64
```

The variable num_clubs has the maximum number of outliers.

```
#Finding how many clubs makes a person an outlier
q3_num_clubs = survey_data.num_clubs.quantile(0.75)
q1_num_clubs = survey_data.num_clubs.quantile(0.25)
print("Tukeys fences = [",q1_num_clubs-1.5*(q3_num_clubs-q1_num_clubs),q3_num_clubs+1.5*(q
```

```
Tukeys fences = [ 0.5 4.5 ]
```

People joining no club, or more than 4 clubs are outliers.

8 Data wrangling

Data wrangling refers to combining, transforming, and re-arranging data to make it suitable for further analysis. We'll use Pandas for all data wrangling operations.

```
import pandas as pd
import numpy as np
import seaborn as sns
```

8.1 Hierarchical indexing

Until now we have seen only a single level of indexing in the rows and columns of a Pandas DataFrame. Hierarchical indexing refers to having multiple index levels on an axis (row / column) of a Pandas DataFrame. It helps us to work with a higher dimensional data in a lower dimensional form.

8.1.1 Hierarchical indexing in Pandas Series

Let us define Pandas Series as we defined in Chapter 5:

```
#Defining a Pandas Series
series_example = pd.Series(['these','are','english','words','estas','son','palabras','en',
                            'ce','sont','des','françai','mots'])
series_example
```

```
0        these
1          are
2      english
3        words
4        estas
5          son
6     palabras
7           en
8      español
```

```
9           ce
10        sont
11         des
12     françai
13        mots
dtype: object
```

Let us use the attribute `nlevels` to find the number of levels of the row indices of this Series:

```
series_example.index.nlevels
```

```
1
```

The Series `series_example` has only one level of row indices.

Let us introduce another level of row indices while defining the Series:

```
#Defining a Pandas Series with multiple levels of row indices
series_example = pd.Series(['these','are','english','words','estas','son','palabras','en',
                            'ce','sont','des','françai','mots'],
                           index=[['English']*4+['Spanish']*5+['French']*5,list(range(1,5))
series_example
```

```
English  1       these
         2         are
         3     english
         4       words
Spanish  1       estas
         2         son
         3    palabras
         4          en
         5     español
French   1          ce
         2        sont
         3         des
         4     françai
         5        mots
dtype: object
```

In the above Series, there are two levels of row indices:

```
series_example.index.nlevels
```

```
2
```

8.1.2 Hierarchical indexing in Pandas DataFrame

In a Pandas DataFrame, both the rows and the columns can have hierarchical indexing. For example, consider the DataFrame below:

```
data=np.array([[771517,2697000,815201,3849000],[4.2,5.6,2.8,4.6],
             [7.8,234.5,46.9,502],[6749, 597, 52, 305]])
df_example = pd.DataFrame(data,index = [['Demographics']*2+['Geography']*2,
                                        ['Population','Unemployment (%)','Area (mile-sq)','E
                    columns = [['Illinois']*2+['California']*2,['Evanston','Chicago','San
df_example
```

		Illinois		California	
		Evanston	Chicago	San Francisco	Los Angeles
Demographics	Population	771517.0	2697000.0	815201.0	3849000.0
	Unemployment (%)	4.2	5.6	2.8	4.6
Geography	Area (mile-sq)	7.8	234.5	46.9	502.0
	Elevation (feet)	6749.0	597.0	52.0	305.0

In the above DataFrame, both the rows and columns have 2 levels of indexing. The number of levels of column indices can be found using the attribute `nlevels`:

```
df_example.columns.nlevels
```

```
2
```

The `columns` attribute will now have a *MultiIndex* datatype in contrast to the *Index* datatype with single level of indexing. The same holds for row indices.

```
type(df_example.columns)
```

```
pandas.core.indexes.multi.MultiIndex
```

```
df_example.columns
```

```
MultiIndex([(  'Illinois',      'Evanston'),
            (  'Illinois',       'Chicago'),
            ('California', 'San Francisco'),
            ('California',   'Los Angeles')],
           )
```

The hierarchical levels can have names. Let us assign names to the each level of the row and column labels:

```
#Naming the row indices levels
df_example.index.names=['Information type', 'Statistic']

#Naming the column indices levels
df_example.columns.names=['State', 'City']

#Viewing the DataFrame
df_example
```

	State	Illinois		California	
	City	Evanston	Chicago	San Francisco	Los Angeles
Information type	Statistic				
Demographics	Population	771517.0	2697000.0	815201.0	3849000.0
	Unemployment (%)	4.2	5.6	2.8	4.6
Geography	Area (mile-sq)	7.8	234.5	46.9	502.0
	Elevation (feet)	6749.0	597.0	52.0	305.0

Observe that the names of the row and column labels appear when we view the DataFrame.

8.1.2.1 `get_level_values()`

The names of the column levels can be obtained using the function `get_level_values()`. The outer-most level corresponds to the *level = 0*, and it increases as we go to the inner levels.

```
#Column levels at level 0 (the outer level)
df_example.columns.get_level_values(0)
```

```
Index(['Illinois', 'Illinois', 'California', 'California'], dtype='object', name='State')
```

```
#Column levels at level 1 (the inner level)
df_example.columns.get_level_values(1)
```

```
Index(['Evanston', 'Chicago', 'San Francisco', 'Los Angeles'], dtype='object', name='City')
```

8.1.3 Subsetting data

We can use the indices at the outer levels to concisely subset a Series / DataFrame.

The first four observations of the Series `series_example` correspond to the outer row index `English`, while the last 5 rows correspond to the outer row index `Spanish`. Let us subset all the observations corresponding to the outer row index `English`:

```
#Subsetting data by row-index
series_example['English']
```

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

Just like in the case of single level indices, if we wish to subset corresponding to multiple outer-level indices, we put the indices within an additional box bracket `[]`. For example, let us subset all the observations corresponding to the row-indices `English` and `French`:

```
#Subsetting data by multiple row-indices
series_example[['English','French']]
```

```
English  1      these
         2        are
         3    english
         4      words
French   1         ce
         2       sont
         3        des
         4    françai
```

```
        5      mots
dtype: object
```

We can also subset data using the inner row index. However, we will need to put a : sign to indicate that the row label at the inner level is being used.

```
#Subsetting data by row-index
series_example[:,2]
```

```
English     are
Spanish     son
French     sont
dtype: object
```

```
#Subsetting data by multiple row-indices
series_example.loc[:,[1,2]]
```

```
English  1    these
Spanish  1    estas
French   1       ce
English  2      are
Spanish  2      son
French   2     sont
dtype: object
```

As in Series, we can concisely subset rows / columns in a DataFrame based on the index at the outer levels.

```
df_example['Illinois']
```

	City	Evanston	Chicago
Information type	Statistic		
Demographics	Population	771517.0	2697000.0
	Unemployment (%)	4.2	5.6
Geography	Area (mile-sq)	7.8	234.5
	Elevation (feet)	6749.0	597.0

Note that the dataype of each column name is a tuple. For example, let us find the datatype of the 1^{st} column name:

```
#First column name
df_example.columns[0]
```

```
('Illinois', 'Evanston')
```

```
#Datatype of first column name
type(df_example.columns[0])
```

```
tuple
```

Thus columns at the inner levels can be accessed by specifying the name as a tuple. For example, let us subset the column `Evanston`:

```
#Subsetting the column 'Evanston'
df_example[('Illinois','Evanston')]
```

```
Information type  Statistic
Demographics      Population           771517.0
                  Unemployment (%)          4.2
Geography         Area (mile-sq)            7.8
                  Elevation (feet)       6749.0
Name: (Illinois, Evanston), dtype: float64
```

```
#Subsetting the columns 'Evanston' and 'Chicago' of the outer column level 'Illinois'
df_example.loc[:,('Illinois',['Evanston','Chicago'])]
```

	State	Illinois	
	City	Evanston	Chicago
Information type	Statistic		
Demographics	Population	771517.0	2697000.0
	Unemployment (%)	4.2	5.6
Geography	Area (mile-sq)	7.8	234.5
	Elevation (feet)	6749.0	597.0

8.1.4 Practice exercise 1

Read the table consisting of GDP per capita of countries from the webpage: https://en.wikipedia.org/wiki/List_o
.

To only read the relevant table, read the tables that contain the word *'Country'*.

8.1.4.1

How many levels of indexing are there in the rows and columns?

```
dfs = pd.read_html('https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)_per_c
gdp_per_capita = dfs[0]
gdp_per_capita.head()
```

	Country/Territory	UN Region	IMF[4]		World Bank[5]		United Nations[6]	
	Country/Territory	UN Region	Estimate	Year	Estimate	Year	Estimate	Year
0	Liechtenstein *	Europe	—	—	169049	2019	180227	2020
1	Monaco *	Europe	—	—	173688	2020	173696	2020
2	Luxembourg *	Europe	127673	2022	135683	2021	117182	2020
3	Bermuda *	Americas	—	—	110870	2021	123945	2020
4	Ireland *	Europe	102217	2022	99152	2021	86251	2020

Just by looking at the DataFrame, it seems as if there are two levels of indexing for columns and one level of indexing for rows. However, let us confirm it with the `nlevels` attribute.

```
gdp_per_capita.columns.nlevels
```

```
2
```

Yes, there are 2 levels of indexing for columns.

```
gdp_per_capita.index.nlevels
```

```
1
```

There is one level of indexing for rows.

8.1.4.2

Subset a DataFrame that selects the country, and the United Nations' estimates of GDP per capita with the corresponding year.

```
gdp_per_capita.loc[:,['Country/Territory','United Nations[6]']]
```

	Country/Territory	United Nations[6]	
	Country/Territory	Estimate	Year
0	Liechtenstein *	180227	2020
1	Monaco *	173696	2020
2	Luxembourg *	117182	2020
3	Bermuda *	123945	2020
4	Ireland *	86251	2020
...	...	...	...
217	Madagascar *	470	2020
218	Central African Republic *	481	2020
219	Sierra Leone *	475	2020
220	South Sudan *	1421	2020
221	Burundi *	286	2020

8.1.4.3

Subset a DataFrame that selects only the World Bank and United Nations' estimates of GDP per capita without the corresponding year or country.

```
gdp_per_capita.loc[:,(['World Bank[5]','United Nations[6]'],'Estimate')]
```

	World Bank[5] Estimate	United Nations[6] Estimate
0	169049	180227
1	173688	173696
2	135683	117182
3	110870	123945
4	99152	86251
...	...	...
217	515	470

	World Bank[5] Estimate	United Nations[6] Estimate
218	512	481
219	516	475
220	1120	1421
221	237	286

8.1.4.4

Subset a DataFrame that selects the country and only the World Bank and United Nations' estimates of GDP per capita without the corresponding year or country.

```
gdp_per_capita.loc[:,[('Country/Territory','Country/Territory'),('United Nations[6]','Esti
```

	Country/Territory Country/Territory	United Nations[6] Estimate	World Bank[5] Estimate
0	Liechtenstein *	180227	169049
1	Monaco *	173696	173688
2	Luxembourg *	117182	135683
3	Bermuda *	123945	110870
4	Ireland *	86251	99152
...	...	...	...
217	Madagascar *	470	515
218	Central African Republic *	481	512
219	Sierra Leone *	475	516
220	South Sudan *	1421	1120
221	Burundi *	286	237

8.1.4.5

Drop all columns consisting of years. Use the `level` argument of the `drop()` method.

```
gdp_per_capita = gdp_per_capita.drop(columns='Year',level=1)
gdp_per_capita
```

	Country/Territory	UN Region	IMF[4]	World Bank[5]	United Nations[6]
	Country/Territory	UN Region	Estimate	Estimate	Estimate
0	Liechtenstein *	Europe	—	169049	180227
1	Monaco *	Europe	—	173688	173696
2	Luxembourg *	Europe	127673	135683	117182
3	Bermuda *	Americas	—	110870	123945
4	Ireland *	Europe	102217	99152	86251
...	...	...	...	...	...
217	Madagascar *	Africa	522	515	470
218	Central African Republic *	Africa	496	512	481
219	Sierra Leone *	Africa	494	516	475
220	South Sudan *	Africa	328	1120	1421
221	Burundi *	Africa	293	237	286

8.1.4.6

In the dataset obtained above, drop the inner level of the column labels. Use the `droplevel()` method.

```
gdp_per_capita = gdp_per_capita.droplevel(1,axis=1)
gdp_per_capita
```

	Country/Territory	UN Region	IMF[4]	World Bank[5]	United Nations[6]
0	Liechtenstein *	Europe	—	169049	180227
1	Monaco *	Europe	—	173688	173696
2	Luxembourg *	Europe	127673	135683	117182
3	Bermuda *	Americas	—	110870	123945
4	Ireland *	Europe	102217	99152	86251
...	...	...	...	...	...
217	Madagascar *	Africa	522	515	470
218	Central African Republic *	Africa	496	512	481
219	Sierra Leone *	Africa	494	516	475
220	South Sudan *	Africa	328	1120	1421
221	Burundi *	Africa	293	237	286

8.1.5 Practice exercise 2

Recall problem 2(e) from assignment 3 on Pandas, where we needed to find the African country that is the closest to country G (Luxembourg) with regard to social indicators.

We will solve the question with the regular way in which we use single level of indexing (as you probably did during this assignment), and see if it is easier to do with hierarchical indexing.

Execute the code below that we used to pre-process data to make it suitable for answering this question.

```
#Pre-processing data - execute this code
social_indicator = pd.read_csv("./Datasets/social_indicator.txt",sep="\t",index_col = 0)
social_indicator.geographic_location = social_indicator.geographic_location.apply(lambda x
social_indicator.rename(columns={'geographic_location':'continent'},inplace=True)
social_indicator = social_indicator.sort_index(axis=1)
social_indicator.drop(columns=['region','contraception'],inplace=True)
```

Below is the code to find the African country that is the closest to country G (Luxembourg) using single level of indexing. Your code in the assignment is probably similar to the one below:

```
#Finding the index of the country G (Luxembourg) that has the maximum GDP per capita
country_max_gdp_position = social_indicator.gdpPerCapita.argmax()

#Scaling the social indicator dataset
social_indicator_scaled = social_indicator.iloc[:,2:].apply(lambda x: (x-x.mean())/(x.std(

#Computing the Manhattan distances of all countries from country G (Luxembourg)
manhattan_distances = (social_indicator_scaled-social_indicator_scaled.iloc[country_max_gd

#Finding the indices of African countries
african_countries_indices = social_indicator.loc[social_indicator.continent=='Africa',:].i

#Filtering the Manhattan distances of African countries from country G (Luxembourg)
manhattan_distances_African = manhattan_distances[african_countries_indices]

#Finding the country among African countries that has the least Manhattan distance to coun
social_indicator.loc[manhattan_distances_African.idxmin(),'country']
```

```
'Reunion'
```

8.1.5.1

Use the method `set_index()` to set `continent` and `country` as hierarchical indices of rows. Find the African country that is the closest to country G (Luxembourg) using this hierarchically indexed data. How many lines will be eliminated from the code above? Which lines will be eliminated?

Hint: Since `continent` and `country` are row indices, you don't need to explicitly find:

1. The row index of country G (Luxembourg),
2. The row indices of African countries.
3. The Manhattan distances for African countries.

```
social_indicator.set_index(['continent','country'],inplace = True)
social_indicator_scaled = social_indicator.apply(lambda x: (x-x.mean())/(x.std()))
manhattan_distances = (social_indicator_scaled-social_indicator_scaled.loc[('Europe','Luxe
manhattan_distances['Africa'].idxmin()
```

```
'Reunion'
```

As we have converted the columns `continent` and `country` to row indices, all the lines of code where we were keeping track of the index of country G, African countries, and Manhattan distances of African countries are eliminated. Three lines of code are eliminated.

Hierarchical indexing relieves us from keeping track of indices, if we set indices that are relatable to our analysis.

8.1.5.2

Use the Pandas DataFrame method `mean()` with the *level* argument to find the mean value of all social indicators for each continent.

```
social_indicator.mean(level=0)
```

	economicActivityFemale	economicActivityMale	gdpPerCapita	illiteracyFemale	illiterac
continent					
Asia	41.592683	79.282927	27796.390244	23.635951	13.7808
Africa	46.732258	79.445161	7127.483871	52.907226	33.6735
Oceania	51.280000	77.953333	14525.666667	9.666667	6.58520
North America	45.238095	77.166667	18609.047619	17.390286	14.6099
South America	42.008333	75.575000	15925.916667	9.991667	6.75000
Europe	52.060000	70.291429	45438.200000	2.308343	1.41354

8.1.6 Practice exercise 3

Let us try to find the areas where NU students lack in diversity. Read *survey_data_clean.csv*. Use hierarchical indexing to classify the columns as follows:

Classify the following variables as *lifestyle*:

```
lifestyle = ['fav_alcohol', 'parties_per_month', 'smoke', 'weed','streaming_platforms', 'm
       'sleep_hours_per_day', 'internet_hours_per_day', 'procrastinator', 'num_clubs','stu
```

Classify the following variables as *personality*:

```
personality = ['introvert_extrovert', 'left_right_brained', 'personality_type',
       'num_insta_followers', 'fav_sport','learning_style','dominant_hand']
```

Classify the following variables as *opinion*:

```
opinion = ['love_first_sight', 'expected_marriage_age',  'expected_starting_salary', 'how_
       'fav_number', 'fav_letter', 'fav_season',   'political_affliation', 'cant_change_ma
       'can_change_math_ability', 'math_is_genetic', 'much_effort_is_lack_of_talent']
```

Classify the following variables as *academic information*:

```
academic_info = ['major', 'num_majors_minors',
       'high_school_GPA', 'NU_GPA', 'school_year','AP_stats', 'used_python_before']
```

Classify the following variables as *demographics*:

```
demographics = [ 'only_child','birth_month',
       'living_location_on_campus', 'age', 'height', 'height_father',
       'height_mother',  'childhood_in_US', 'gender', 'region_of_residence']
```

Write a function that finds the number of variables having outliers in a dataset. Apply the function to each of the 5 categories of variables in the dataset. Our hypothesis is that the category that has the maximum number of variables with outliers has the least amount of diversity. For continuous variables, use Tukey's fences criterion to identify outliers. For categorical variables, consider levels having less than 1% observations as outliers. Assume that numeric variables that have more than 2 distinct values are continuous.

Solution:

```python
#Using hierarchical indexing to classify columns

#Reading data
survey_data = pd.read_csv('./Datasets/survey_data_clean.csv')

#Arranging columns in the order of categories
survey_data_HI = survey_data[lifestyle+personality+opinion+academic_info+demographics]

#Creating hierarchical indexing to classify columns
survey_data_HI.columns=[['lifestyle']*len(lifestyle)+['personality']*len(personality)+['op
                       ['academic_info']*len(academic_info)+['demographics']*len(demograph
                        personality+opinion+academic_info+demographics]
```

```python
#Function to identify outliers based on Tukey's fences for continous variables and 1% crit
def rem_outliers(x):
    if ((len(x.value_counts())>2) & (x.dtype!='O')):#continuous variable
        q1 =x.quantile(0.25)
        q3 = x.quantile(0.75)
        intQ_range = q3-q1

        #Tukey's fences
        Lower_fence = q1 - 1.5*intQ_range
        Upper_fence = q3 + 1.5*intQ_range

        num_outliers = ((x<Lower_fence) | (x>Upper_fence)).sum()
        if num_outliers>0:
            return True
        return False
    else:                                       #categorical variable
        if np.min(x.value_counts()/len(x))<0.01:
            return True
        return False
```

```python
#Number of variables containing outlier(s) in each category
for category in survey_data_HI.columns.get_level_values(0).unique():
    print("Number of missing values for category ",category," = ",survey_data_HI[category]
```

```
Number of missing values for category  lifestyle  =  7
Number of missing values for category  personality  =  2
Number of missing values for category  opinion  =  4
Number of missing values for category  academic_info  =  3
```

```
Number of missing values for category  demographics  =  4
```

The *lifestyle* category has the highest number of variables containing outlier(s). If the hypothesis is true, then NU students have the least diversity in their lifestyle, among all the categories.

Although one may say that the *lifestyle* category has the the highest number of columns (as shown below), the proportion of columns having outlier(s) is also the highest for this category.

```
for category in survey_data_HI.columns.get_level_values(0).unique():
    print("Number of columns in category ",category," = ",survey_data_HI[category].shape[1
```

```
Number of columns in category  lifestyle  =  12
Number of columns in category  personality  =  7
Number of columns in category  opinion  =  12
Number of columns in category  academic_info  =  7
Number of columns in category  demographics  =  10
```

8.1.7 Reshaping data

Apart from ease in subsetting data, hierarchical indexing also plays a role in reshaping data.

8.1.7.1 `unstack()` (Pandas Series method)

The Pandas Series method `unstack()` pivots the desired level of row indices to columns, thereby creating a DataFrame. By default, the inner-most level of the row labels is pivoted.

```
#Pivoting the inner-most Series row index to column labels
series_example_unstack = series_example.unstack()
series_example_unstack
```

	1	2	3	4	5
English	these	are	english	words	NaN
French	ce	sont	des	françai	mots
Spanish	estas	son	palabras	en	español

We can pivot the outer level of the row labels by specifying it in the `level` argument:

```
#Pivoting the outer row indices to column labels
series_example_unstack = series_example.unstack(level=0)
series_example_unstack
```

	English	French	Spanish
1	these	ce	estas
2	are	sont	son
3	english	des	palabras
4	words	françai	en
5	NaN	mots	español

8.1.7.2 `unstack()` (Pandas DataFrame method)

The Pandas DataFrame method `unstack()` pivots the specified level of row indices to the new inner-most level of column labels. By default, the inner-most level of the row labels is pivoted.

```
#Pivoting the inner level of row labels to the inner-most level of column labels
df_example.unstack()
```

State	Illinois					
City	Evanston				Chicago	
Statistic	Area (mile-sq)	Elevation (feet)	Population	Unemployement (%)	Area (mile-sq)	Ele
Information type						
Demographics	NaN	NaN	771517.0	4.2	NaN	Na
Geography	7.8	6749.0	NaN	NaN	234.5	597

As with Series, we can pivot the outer level of the row labels by specifying it in the `level` argument:

```
#Pivoting the outer level (level = 0) of row labels to the inner-most level of column labe
df_example.unstack(level=0)
```

State	Illinois				California	
City	Evanston		Chicago		San Francisco	
Information type	Demographics	Geography	Demographics	Geography	Demographics	Geography
Statistic						
Area (mile-sq)	NaN	7.8	NaN	234.5	NaN	46.9
Elevation (feet)	NaN	6749.0	NaN	597.0	NaN	52.0
Population	771517.0	NaN	2697000.0	NaN	815201.0	NaN
Unemployement (%)	4.2	NaN	5.6	NaN	2.8	NaN

8.1.7.3 `stack()`

The inverse of `unstack()` is the `stack()` method, which creates the inner-most level of row indices by pivoting the column labels of the prescribed level.

Note that if the column labels have only one level, we don't need to specify a level.

```
#Stacking the columns of a DataFrame
series_example_unstack.stack()
```

```
English  1       these
         2         are
         3     english
         4       words
French   1          ce
         2        sont
         3         des
         4     françai
         5        mots
Spanish  1       estas
         2         son
         3    palabras
         4          en
         5     español
dtype: object
```

However, if the columns have multiple levels, we can specify the level to stack as the inner-most row level. By default, the inner-most column level is stacked.

```
#Stacking the inner-most column labels inner-most row indices
df_example.stack()
```

Information type	Statistic	State City	California	Illinois
Demographics	Population	Chicago	NaN	2697000.0
		Evanston	NaN	771517.0
		Los Angeles	3849000.0	NaN
		San Francisco	815201.0	NaN
	Unemployement (%)	Chicago	NaN	5.6
		Evanston	NaN	4.2
		Los Angeles	4.6	NaN
		San Francisco	2.8	NaN
Geography	Area (mile-sq)	Chicago	NaN	234.5
		Evanston	NaN	7.8
		Los Angeles	502.0	NaN
		San Francisco	46.9	NaN
	Elevation (feet)	Chicago	NaN	597.0
		Evanston	NaN	6749.0
		Los Angeles	305.0	NaN
		San Francisco	52.0	NaN

```
#Stacking the outer column labels inner-most row indices
df_example.stack(level=0)
```

Information type	Statistic	City State	Chicago	Evanston	Los Angeles	San Francisco
Demographics	Population	California	NaN	NaN	3849000.0	815201.0
		Illinois	2697000.0	771517.0	NaN	NaN
	Unemployement (%)	California	NaN	NaN	4.6	2.8
		Illinois	5.6	4.2	NaN	NaN
Geography	Area (mile-sq)	California	NaN	NaN	502.0	46.9
		Illinois	234.5	7.8	NaN	NaN
	Elevation (feet)	California	NaN	NaN	305.0	52.0
		Illinois	597.0	6749.0	NaN	NaN

8.2 Merging data

The Pandas DataFrame method merge() uses columns defined as *key* column(s) to merge two datasets. In case the *key* column(s) are not defined, the overlapping column(s) are considered as the *key* columns.

8.2.1 Join types

When a dataset is merged with another based on *key* column(s), one of the following four types of join will occur depending on the repetition of the values of the key(s) in the datasets.

(i) One-to-one, (ii) Many-to-one, (iii) One-to-Many, and (iv) Many-to-many

The type of join may sometimes determine the number of rows to be obtained in the merged dataset. If we don't get the expected number of rows in the merged dataset, an investigation of the datsets may be neccessary to identify and resolve the issue. There may be several possible issues, for example, the dataset may not be arranged in a way that we have assumed it to be arranged.

We'll use toy datasets to understand the above types of joins. The *.csv* files with the prefix *student* consist of the names of a few students along with their majors, and the files with the prefix *skills* consist of the names of majors along with the skills imparted by the respective majors.

```
data_student = pd.read_csv('./Datasets/student_one.csv')
data_skill = pd.read_csv('./Datasets/skills_one.csv')
```

8.2.1.1 One-to-one join

Each row in one dataset is linked (or related) to a single row in another dataset based on the key column(s).

```
data_student
```

	Student	Major
0	Kitana	Statistics
1	Jax	Computer Science
2	Sonya	Material Science
3	Johnny	Music

```
data_skill
```

	Major	Skills
0	Statistics	Inference
1	Computer Science	Machine learning
2	Material Science	Structure prediction
3	Music	Opera

```
pd.merge(data_student,data_skill)
```

	Student	Major	Skills
0	Kitana	Statistics	Inference
1	Jax	Computer Science	Machine learning
2	Sonya	Material Science	Structure prediction
3	Johnny	Music	Opera

8.2.1.2 Many-to-one join

One or more rows in one dataset is linked (or related) to a single row in another dataset based on the key column(s).

```
data_student = pd.read_csv('./Datasets/student_many.csv')
data_skill = pd.read_csv('./Datasets/skills_one.csv')
```

```
data_student
```

	Student	Major
0	Kitana	Statistics
1	Kitana	Computer Science
2	Jax	Computer Science
3	Sonya	Material Science
4	Johnny	Music
5	Johnny	Statistics

```
data_skill
```

	Major	Skills
0	Statistics	Inference
1	Computer Science	Machine learning
2	Material Science	Structure prediction
3	Music	Opera

```
pd.merge(data_student,data_skill)
```

	Student	Major	Skills
0	Kitana	Statistics	Inference
1	Johnny	Statistics	Inference
2	Kitana	Computer Science	Machine learning
3	Jax	Computer Science	Machine learning
4	Sonya	Material Science	Structure prediction
5	Johnny	Music	Opera

8.2.1.3 One-to-many join

Each row in one dataset is linked (or related) to one, or more rows in another dataset based on the key column(s).

```
data_student = pd.read_csv('./Datasets/student_one.csv')
data_skill = pd.read_csv('./Datasets/skills_many.csv')
```

```
data_student
```

	Student	Major
0	Kitana	Statistics
1	Jax	Computer Science
2	Sonya	Material Science
3	Johnny	Music

```
data_skill
```

	Major	Skills
0	Statistics	Inference
1	Statistics	Modeling
2	Computer Science	Machine learning
3	Computer Science	Computing
4	Material Science	Structure prediction
5	Music	Opera
6	Music	Pop
7	Music	Classical

```
pd.merge(data_student,data_skill)
```

	Student	Major	Skills
0	Kitana	Statistics	Inference
1	Kitana	Statistics	Modeling
2	Jax	Computer Science	Machine learning
3	Jax	Computer Science	Computing
4	Sonya	Material Science	Structure prediction
5	Johnny	Music	Opera
6	Johnny	Music	Pop
7	Johnny	Music	Classical

8.2.1.4 Many-to-many join

One, or more, rows in one dataset is linked (or related) to one, or more, rows in another dataset using the key column(s).

```
data_student = pd.read_csv('./Datasets/student_many.csv')
data_skill = pd.read_csv('./Datasets/skills_many.csv')
```

```
data_student
```

	Student	Major
0	Kitana	Statistics
1	Kitana	Computer Science
2	Jax	Computer Science
3	Sonya	Material Science

	Student	Major
4	Johnny	Music
5	Johnny	Statistics

```
data_skill
```

	Major	Skills
0	Statistics	Inference
1	Statistics	Modeling
2	Computer Science	Machine learning
3	Computer Science	Computing
4	Material Science	Structure prediction
5	Music	Opera
6	Music	Pop
7	Music	Classical

```
pd.merge(data_student,data_skill)
```

	Student	Major	Skills
0	Kitana	Statistics	Inference
1	Kitana	Statistics	Modeling
2	Johnny	Statistics	Inference
3	Johnny	Statistics	Modeling
4	Kitana	Computer Science	Machine learning
5	Kitana	Computer Science	Computing
6	Jax	Computer Science	Machine learning
7	Jax	Computer Science	Computing
8	Sonya	Material Science	Structure prediction
9	Johnny	Music	Opera
10	Johnny	Music	Pop
11	Johnny	Music	Classical

Note that there are two '*Statistics*' rows in `data_student`, and two '*Statistics*' rows in `data_skill`, resulting in 2x2 = 4 '*Statistics*' rows in the merged data. The same is true for the '*Computer Science*' `Major`.

8.2.2 Join types with `how` argument

The above mentioned types of join (one-to-one, many-to-one, etc.) occur depening on the structure of the datasets being merged. We don't have control over the type of join. However, we can control `how` the joins are occurring. We can merge (or *join*) two datasets in one of the following four ways:

(i) `inner` join, (ii) `left` join, (iii) `right` join, (iv) `outer` join

8.2.2.1 inner join

This is the join that occurs by default, i.e., without specifying the `how` argument in the `merge()` function. In `inner` join, only those observations are merged that have the same value(s) in the *key* column(s) of both the datasets.

```
data_student = pd.read_csv('./Datasets/student_how.csv')
data_skill = pd.read_csv('./Datasets/skills_how.csv')
```

```
data_student
```

	Student	Major
0	Kitana	Statistics
1	Jax	Computer Science
2	Sonya	Material Science

```
data_skill
```

	Major	Skills
0	Statistics	Inference
1	Computer Science	Machine learning
2	Music	Opera

```
pd.merge(data_student,data_skill)
```

	Student	Major	Skills
0	Kitana	Statistics	Inference

	Student	Major	Skills
1	Jax	Computer Science	Machine learning

When you may use inner join? You should use inner join when you cannot carry out the analysis unless the observation corresponding to the *key column(s)* is present in both the tables.

Example: Suppose you wish to analyze the association between vaccinations and covid infection rate based on country-level data. In one of the datasets, you have the infection rate for each country, while in the other one you have the number of vaccinations in each country. The countries which have either the vaccination or the infection rate missing, cannot help analyze the association. In such as case you may be interested only in countries that have values for both the variables. Thus, you will use inner join to discard the countries with either value missing.

8.2.2.2 left join

In `left` join, the merged dataset will have all the rows of the dataset that is specified first in the `merge()` function. Only those observations of the other dataset will be merged whose value(s) in the *key* column(s) exist in the dataset specified first in the `merge()` function.

```
pd.merge(data_student,data_skill,how='left')
```

	Student	Major	Skills
0	Kitana	Statistics	Inference
1	Jax	Computer Science	Machine learning
2	Sonya	Material Science	NaN

When you may use left join? You should use left join when the primary variable(s) of interest are present in the one of the datasets, and whose missing values cannot be imputed. The variable(s) in the other dataset may not be as important or it may be possible to reasonably impute their values, if missing corresponding to the observation in the primary dataset.

Examples:

1) Suppose you wish to analyze the association between the covid infection rate and the government effectiveness score (a metric used to determine the effectiveness of the government in implementing policies, upholding law and order etc.) based on the data of all countries. Let us say that one of the datasets contains the covid infection rate, while the other one contains the government effectiveness score for each country. If the infection

rate for a country is missing, it might be hard to impute. However, the government effectiveness score may be easier to impute based on GDP per capita, crime rate etc. - information that is easily available online. In such a case, you may wish to use a left join where you keep all the countries for which the infection rate is known.

2) Suppose you wish to analyze the association between demographics such as age, income etc. and the amount of credit card spend. Let us say one of the datasets contains the demographic information of each customer, while the other one contains the credit card spend for the customers who made at least one purchase. In such as case, you may want to do a left join as customers not making any purchase might be absent in the card spend data. Their spend can be imputed as zero after merging the datasets.

8.2.2.3 right join

In `right` join, the merged dataset will have all the rows of the dataset that is specified second in the `merge()` function. Only those observations of the other dataset will be merged whose value(s) in the *key* column(s) exist in the dataset specified second in the `merge()` function.

```
pd.merge(data_student,data_skill,how='right')
```

	Student	Major	Skills
0	Kitana	Statistics	Inference
1	Jax	Computer Science	Machine learning
2	NaN	Music	Opera

When you may use right join? You can always use a left join instead of a right join. Their purpose is the same.

8.2.2.4 outer join

In `outer` join, the merged dataset will have all the rows of both the datasets being merged.

```
pd.merge(data_student,data_skill,how='outer')
```

	Student	Major	Skills
0	Kitana	Statistics	Inference
1	Jax	Computer Science	Machine learning
2	Sonya	Material Science	NaN
3	NaN	Music	Opera

Student	Major	Skills

When you may use outer join? You should use an outer join when you cannot afford to lose data present in either of the tables. All the other joins may result in loss of data.

Example: Suppose I took two course surveys for this course. If I need to analyze student sentiment during the course, I will take an outer join of both the surveys. Assume that each survey is a dataset, where each row corresponds to a unique student. Even if a student has answered one of the two surverys, it will be indicative of the sentiment, and will be useful to keep in the merged dataset.

8.3 Concatenating datasets

The Pandas DataFrame method `concat()` is used to stack datasets along an axis. The method is similar to NumPy's `concatenate()` method.

Example: You are given the life expectancy data of each continent as a separate *.csv file. Visualize the change of life expectancy over time for different continents.

```
data_asia = pd.read_csv('./Datasets/gdp_lifeExpec_Asia.csv')
data_europe = pd.read_csv('./Datasets/gdp_lifeExpec_Europe.csv')
data_africa = pd.read_csv('./Datasets/gdp_lifeExpec_Africa.csv')
data_oceania = pd.read_csv('./Datasets/gdp_lifeExpec_Oceania.csv')
data_americas = pd.read_csv('./Datasets/gdp_lifeExpec_Americas.csv')
```

```
#Appending all the data files, i.e., stacking them on top of each other
data_all_continents = pd.concat([data_asia,data_europe,data_africa,data_oceania,data_ameri
data_all_continents
```

		country	year	lifeExp	pop	gdpPercap
	0	Afghanistan	1952	28.801	8425333	779.445314
	1	Afghanistan	1957	30.332	9240934	820.853030
Asia	2	Afghanistan	1962	31.997	10267083	853.100710
	3	Afghanistan	1967	34.020	11537966	836.197138
	4	Afghanistan	1972	36.088	13079460	739.981106
...	...	...	...	...	...	...
	295	Venezuela	1987	70.190	17910182	9883.584648
	296	Venezuela	1992	71.150	20265563	10733.926310
Americas	297	Venezuela	1997	72.146	22374398	10165.495180

	country	year	lifeExp	pop	gdpPercap
298	Venezuela	2002	72.766	24287670	8605.047831
299	Venezuela	2007	73.747	26084662	11415.805690

Let's have the continent as a column as we need to use that in the visualization.

```
data_all_continents.reset_index(inplace = True)
```

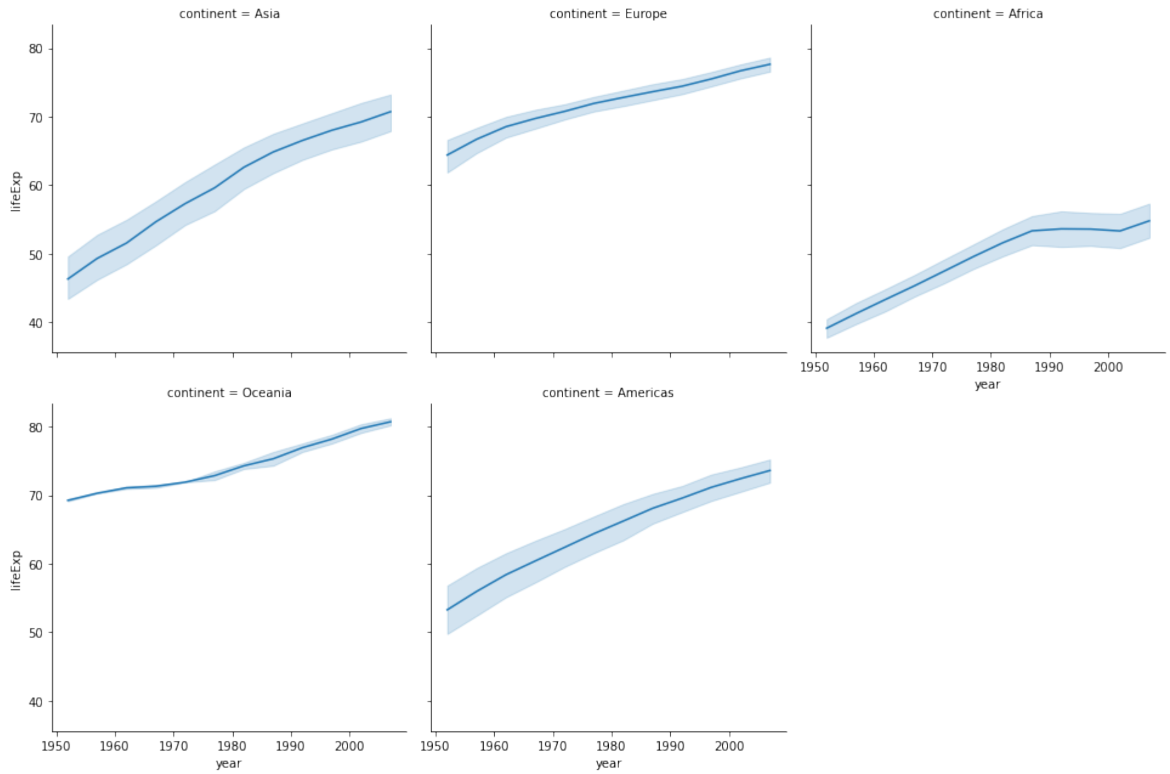
```
data_all_continents.head()
```

	level_0	level_1	country	year	lifeExp	pop	gdpPercap
0	Asia	0	Afghanistan	1952	28.801	8425333	779.445314
1	Asia	1	Afghanistan	1957	30.332	9240934	820.853030
2	Asia	2	Afghanistan	1962	31.997	10267083	853.100710
3	Asia	3	Afghanistan	1967	34.020	11537966	836.197138
4	Asia	4	Afghanistan	1972	36.088	13079460	739.981106

```
data_all_continents.drop(columns = 'level_1',inplace = True)
data_all_continents.rename(columns = {'level_0':'continent'},inplace = True)
data_all_continents.head()
```

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

```
#change of life expectancy over time for different continents
a = sns.FacetGrid(data_all_continents,col = 'continent',col_wrap = 3,height = 4.5,aspect =
a.map(sns.lineplot,'year','lifeExp')
a.add_legend()
```



In the above example, datasets were appended (or stacked on top of each other).

Datasets can also be concatenated side-by-side (by providing the argument *axis = 1* with the *concat()* function) as we saw with the *merge* function.

8.3.1 Practice exercise 4

Read the documentations of the Pandas DataFrame methods `merge()` and `concat()`, and identify the differences. Mention examples when you can use (i) either, (ii) only `concat()`, (iii) only `merge()`

Solution:

(i) If we need to merge datasets using row indices, we can use either function.

(ii) If we need to stack datasets one on top of the other, we can only use `concat()`

(iii) If we need to merge datasets using overlapping columns we can only use `merge()`

8.4 Reshaping data

Data often needs to be re-arranged to ease analysis.

8.4.1 Pivoting "long" to "wide" format

pivot()

This function helps re-arrange data from the *'long'* form to a *'wide'* form.

Example: Let us consider the dataset `data_all_continents` obtained in the previous section after concatenating the data of all the continents.

```
data_all_continents.head()
```

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

8.4.1.1 Pivoting a single column

For visualizing life expectancy in 2007 against life expectancy in 1957, we will need to filter the data, and then make the plot. Everytime that we need to compare a metric for a year against another year, we will need to filter the data.

If we need to often compare metrics of a year against another year, it will be easier to have each year as a separate column, instead of having all years in a single column.

As we are increasing the number of columns and decreasing the number of rows, we are re-arranging the data from *long*-form to *wide*-form.

```
data_wide = data_all_continents.pivot(index = ['continent','country'],columns = 'year',val
```

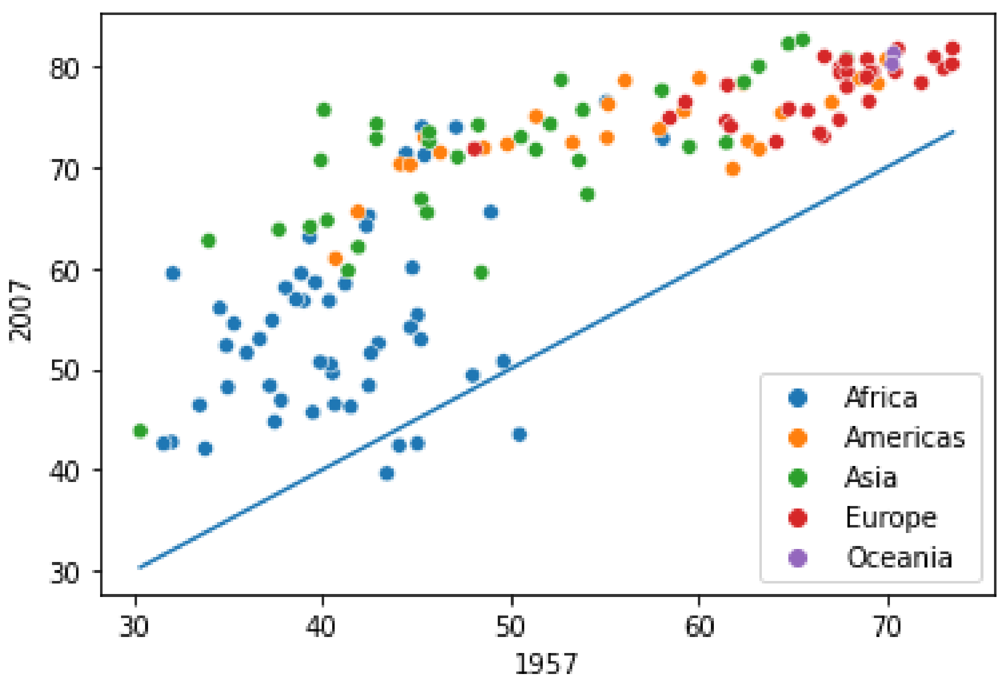
```
data_wide.head()
```

continent	year country	1952	1957	1962	1967	1972	1977	1982	1987	1992	199
Africa	Algeria	43.077	45.685	48.303	51.407	54.518	58.014	61.368	65.799	67.744	69.1
	Angola	30.015	31.999	34.000	35.985	37.928	39.483	39.942	39.906	40.647	40.9
	Benin	38.223	40.358	42.618	44.885	47.014	49.190	50.904	52.337	53.919	54.7
	Botswana	47.622	49.618	51.520	53.298	56.024	59.319	61.484	63.622	62.745	52.5
	Burkina Faso	31.975	34.906	37.814	40.697	43.591	46.137	48.122	49.557	50.260	50.3

With values of `year` as columns, it is easy to compare any metric for different years.

```
#visualizing the change in life expectancy of all countries in 2007 as compared to that in
sns.scatterplot(data = data_wide, x = 1957,y=2007,hue = 'continent')
sns.lineplot(data = data_wide, x = 1957,y = 1957)
```

```
<AxesSubplot:xlabel='1957', ylabel='2007'>
```



Observe that for some African countries, the life expectancy has decreased after 50 years. It is worth investigating these countries to identify factors associated with the decrease.

8.4.1.2 Pivoting multiple columns

In the above transformation, we retained only `lifeExp` in the *'wide'* dataset. Suppose, we are also interested in visualizing GDP per capita of countries in one year against another year. In that case, we must have `gdpPercap` in the *'wide'*-form data as well.

Let us create a dataset named as `data_wide_lifeExp_gdpPercap` that will contain both `lifeExp` and `gdpPercap` for each year in a separate column. We will specify the columns to pivot in the *values* argument of the `pivot()` function.

```
data_wide_lifeExp_gdpPercap = data_all_continents.pivot(index = ['continent','country'],co
data_wide_lifeExp_gdpPercap.head()
```

		lifeExp									
	year	1952	1957	1962	1967	1972	1977	1982	1987	1992	199
continent	country										
Africa	Algeria	43.077	45.685	48.303	51.407	54.518	58.014	61.368	65.799	67.744	69.1
	Angola	30.015	31.999	34.000	35.985	37.928	39.483	39.942	39.906	40.647	40.9
	Benin	38.223	40.358	42.618	44.885	47.014	49.190	50.904	52.337	53.919	54.7
	Botswana	47.622	49.618	51.520	53.298	56.024	59.319	61.484	63.622	62.745	52.5
	Burkina Faso	31.975	34.906	37.814	40.697	43.591	46.137	48.122	49.557	50.260	50.3

The metric for each year is now in a separate column, and can be visualized directly. Note that re-arranging the dataset from the *'long'*-form to *'wide-form'* leads to hierarchical indexing of columns when multiple *'values'* need to be re-arranged. In this case, the multiple *'values'* that need to be re-arranged are `lifeExp` and `gdpPercap`.

8.4.2 Melting "wide" to "long" format

`melt()`

This function is used to re-arrange the dataset from the *'wide'* form to the *'long'* form.

8.4.2.1 Melting columns with a single type of value

Let us consider `data_wide` created in the previous section.

```
data_wide.head()
```

continent	year country	1952	1957	1962	1967	1972	1977	1982	1987	1992	199
Africa	Algeria	43.077	45.685	48.303	51.407	54.518	58.014	61.368	65.799	67.744	69.1
	Angola	30.015	31.999	34.000	35.985	37.928	39.483	39.942	39.906	40.647	40.9
	Benin	38.223	40.358	42.618	44.885	47.014	49.190	50.904	52.337	53.919	54.7
	Botswana	47.622	49.618	51.520	53.298	56.024	59.319	61.484	63.622	62.745	52.5
	Burkina Faso	31.975	34.906	37.814	40.697	43.591	46.137	48.122	49.557	50.260	50.3

Suppose, we wish to visualize the change of life expectancy over time for different continents, as we did in section 8.3. For plotting `lifeExp` against `year`, all the years must be in a single column. Thus, we need to *melt* the columns of `data_wide` to a single column and call it `year`.

But before melting the columns in the above dataset, we will convert `continent` to a column, as we need to make subplots based on continent.

The Pandas DataFrame method `reset_index()` can be used to remove one or more levels of indexing from the DataFrame.

```
#Making 'continent' a column instead of row-index at level 0
data_wide.reset_index(inplace=True,level=0)
data_wide.head()
```

year country	continent	1952	1957	1962	1967	1972	1977	1982	1987	1992	199
Algeria	Africa	43.077	45.685	48.303	51.407	54.518	58.014	61.368	65.799	67.744	69.1
Angola	Africa	30.015	31.999	34.000	35.985	37.928	39.483	39.942	39.906	40.647	40.9
Benin	Africa	38.223	40.358	42.618	44.885	47.014	49.190	50.904	52.337	53.919	54.7
Botswana	Africa	47.622	49.618	51.520	53.298	56.024	59.319	61.484	63.622	62.745	52.5
Burkina Faso	Africa	31.975	34.906	37.814	40.697	43.591	46.137	48.122	49.557	50.260	50.3

```
data_melted=pd.melt(data_wide,id_vars = ['continent'],var_name = 'Year',value_name = 'Life
data_melted.head()
```

	continent	Year	LifeExp
0	Africa	1952	43.077
1	Africa	1952	30.015
2	Africa	1952	38.223
3	Africa	1952	47.622

	continent	Year	LifeExp
4	Africa	1952	31.975

With the above DataFrame, we can visualize the mean life expectancy against year separately for each continent.

If we wish to have country also in the above data, we can keep it while resetting the index:

```
#Creating 'data_wide' again
data_wide = data_all_continents.pivot(index = ['continent','country'],columns = 'year',val

#Resetting the row-indices to default values
data_wide.reset_index(inplace=True)
data_wide.head()
```

year	continent	country	1952	1957	1962	1967	1972	1977	1982	1987	1992
0	Africa	Algeria	43.077	45.685	48.303	51.407	54.518	58.014	61.368	65.799	67.74
1	Africa	Angola	30.015	31.999	34.000	35.985	37.928	39.483	39.942	39.906	40.64
2	Africa	Benin	38.223	40.358	42.618	44.885	47.014	49.190	50.904	52.337	53.91
3	Africa	Botswana	47.622	49.618	51.520	53.298	56.024	59.319	61.484	63.622	62.74
4	Africa	Burkina Faso	31.975	34.906	37.814	40.697	43.591	46.137	48.122	49.557	50.26

```
#Melting the 'year' column
data_melted=pd.melt(data_wide,id_vars = ['continent','country'],var_name = 'Year',value_na
data_melted.head()
```

	continent	country	Year	LifeExp
0	Africa	Algeria	1952	43.077
1	Africa	Angola	1952	30.015
2	Africa	Benin	1952	38.223
3	Africa	Botswana	1952	47.622
4	Africa	Burkina Faso	1952	31.975

8.4.2.2 Melting columns with multiple types of values

Consider the dataset created in Section 8.4.1.2. It has two types of values - `lifeExp` and `gdpPercapita`, which are the column labels at the outer level. The `melt()` function will melt all the years of data into a single column. However, it will create another column based on the outer level column labels - `lifeExp` and `gdpPercapita` to distinguish between these two types of values. Here, we see that the function `melt()` internally uses hierarchical indexing to handle the transformation of multiple types of columns.

```
data_melt = pd.melt(data_wide_lifeExp_gdpPercap.reset_index(),id_vars = ['continent','coun
data_melt.head()
```

	continent	country	Metric	year	value
0	Africa	Algeria	lifeExp	1952	43.077
1	Africa	Angola	lifeExp	1952	30.015
2	Africa	Benin	lifeExp	1952	38.223
3	Africa	Botswana	lifeExp	1952	47.622
4	Africa	Burkina Faso	lifeExp	1952	31.975

Although the data above is in *'long'*-form, it is not quiet in its original format, as in `data_all_continents`. We need to pivot again by `Metric` to have two separate columns of `gdpPercap` and `lifeExp`.

```
data_restore = data_melt.pivot(index = ['continent','country','year'],columns = 'Metric')
data_restore.head()
```

			value	
		Metric	gdpPercap	lifeExp
continent	country	year		
Africa	Algeria	1952	2449.008185	43.077
		1957	3013.976023	45.685
		1962	2550.816880	48.303
		1967	3246.991771	51.407
		1972	4182.663766	54.518

Now, we can convert the row indices of `continent` and `country` to columns to restore the dataset to the same form as `data_all_continents`.

```
data_restore.reset_index(inplace = True)
data_restore.head()
```

	continent	country	year	value	
Metric				gdpPercap	lifeExp
0	Africa	Algeria	1952	2449.008185	43.077
1	Africa	Algeria	1957	3013.976023	45.685
2	Africa	Algeria	1962	2550.816880	48.303
3	Africa	Algeria	1967	3246.991771	51.407
4	Africa	Algeria	1972	4182.663766	54.518

8.4.3 Practice exercise 5

8.4.3.1

Both `unstack()` and `pivot()` seem to transform the data from the *'long'* form to the *'wide'* form. Is there a difference between the two functions?

Solution:

Yes, both the functions transform the data from the *'long'* form to the *'wide'* form. However, `unstack()` pivots the row indices, while `pivot()` pivots the columns of the DataFrame.

Even though both functions are a bit different, it is possible to just use one of them to perform a reshaping operation. If we wish to pivot a column, we can either use `pivot()` directly on the column, or we can convert the column to row indices and then use `unstack()`. If we wish to pivot row indices, we can either use `unstack()` directly on the row indices, or we can convert row indices to a column and then use `pivot()`.

To summarise, using one function may be more straightforward than using the other one, but either can be used for reshaping data from the *'long'* form to the *'wide'* form.

Below is an example where we perform the same reshaping operation with either function.

Consider the data `data_all_continent`. Suppose we wish to transform it to `data_wide` as we did using `pivot()` in Section 8.4.1.1. Let us do it using `unstack()`, instead of `pivot()`.

The first step will be to reindex data to set `year` as row indices, and also `continent` and `country` as row indices because these two column were set as indices with the `pivot()` function in Section 8.4.1.1.

```
#Reindexing data to make 'continent', 'country', and 'year' as hierarchical row indices
data_reindexed=data_all_continents.set_index(['continent','country','year'])
data_reindexed
```

			lifeExp	pop	gdpPercap
continent	country	year			
		1952	28.801	8425333	779.445314
		1957	30.332	9240934	820.853030
Asia	Afghanistan	1962	31.997	10267083	853.100710
		1967	34.020	11537966	836.197138
		1972	36.088	13079460	739.981106
...	...	...	...	...	...
		1987	70.190	17910182	9883.584648
		1992	71.150	20265563	10733.926310
Americas	Venezuela	1997	72.146	22374398	10165.495180
		2002	72.766	24287670	8605.047831
		2007	73.747	26084662	11415.805690

Now we can use **`unstack()`** to pivot the desired row index, i.e., **`year`**. Also, since we are only interested in pivoting the values of **`lifeExp`** (as in the example in Section 8.4.1.1), we will filter the pivoted data with the **`lifeExp`** column label.

```
data_wide_with_unstack=data_reindexed.unstack('year')['lifeExp']
data_wide_with_unstack
```

	year	1952	1957	1962	1967	1972	1977	1982	1987	1992
continent	country									
	Algeria	43.077	45.685	48.303	51.407	54.518	58.014	61.368	65.799	67.744
	Angola	30.015	31.999	34.000	35.985	37.928	39.483	39.942	39.906	40.647
Africa	Benin	38.223	40.358	42.618	44.885	47.014	49.190	50.904	52.337	53.919
	Botswana	47.622	49.618	51.520	53.298	56.024	59.319	61.484	63.622	62.745
	Burkina Faso	31.975	34.906	37.814	40.697	43.591	46.137	48.122	49.557	50.260
...	...	...	...	...	...	...	...	...	...	...
	Switzerland	69.620	70.560	71.320	72.770	73.780	75.390	76.210	77.410	78.030
Europe	Turkey	43.585	48.079	52.098	54.336	57.005	59.507	61.036	63.108	66.146
	United Kingdom	69.180	70.420	70.760	71.360	72.010	72.760	74.040	75.007	76.420
Oceania	Australia	69.120	70.330	70.930	71.100	71.930	73.490	74.740	76.320	77.560
	New Zealand	69.390	70.260	71.240	71.520	71.890	72.220	73.840	74.320	76.330

The above dataset is the same as that obtained using the **`pivot()`** function in Section 8.4.1.1.

8.4.3.2

Both `stack()` and `melt()` seem to transform the data from the *'wide'* form to the *'long'* form. Is there a difference between the two functions?

Solution:

Following the trend of the previous question, we can always use `stack()` instead of `melt()` and vice-versa. The main difference is that `melt()` lets us choose the indentifier columns with the argument `id_vars`. However, if we use `stack()`, we will need to set the relevant melted row indices as columns. On the other hand, if we wished to have the melted columns as row indices, we can either directly use `stack()` or use `melt()` and then set the desired columns as row indices.

To summarise, using one function may be more straightforward than using the other one, but either can be used for reshaping data from the *'wide'* form to the *'long'* form.

Let us melt the data `data_wide_with_unstack` using the `stack()` function to obtain the same dataset as obtained with the `melt()` function in Section 8.4.1.2.

```
#Stacking the data
data_stacked = data_wide_with_unstack.stack()
data_stacked
```

```
continent  country      year
Africa     Algeria      1952    43.077
                        1957    45.685
                        1962    48.303
                        1967    51.407
                        1972    54.518
                                 ...
Oceania    New Zealand  1987    74.320
                        1992    76.330
                        1997    77.550
                        2002    79.110
                        2007    80.204
Length: 1704, dtype: float64
```

Now we need to convert the row indices `continent` and `country` to columns as in the melted data in Section 8.4.1.2.

```
#Putting 'continent' and 'country' as columns
data_long_with_stack = data_stacked.reset_index()
```

```
data_long_with_stack
```

	continent	country	year	0
0	Africa	Algeria	1952	43.077
1	Africa	Algeria	1957	45.685
2	Africa	Algeria	1962	48.303
3	Africa	Algeria	1967	51.407
4	Africa	Algeria	1972	54.518
...	...	...	...	...
1699	Oceania	New Zealand	1987	74.320
1700	Oceania	New Zealand	1992	76.330
1701	Oceania	New Zealand	1997	77.550
1702	Oceania	New Zealand	2002	79.110
1703	Oceania	New Zealand	2007	80.204

Finally, we need to rename the column named as 0 to `LifeExp` to obtain the same dataset as in Section 8.4.1.2.

```
#Renaming column 0 to 'LifeExp'
data_long_with_stack.rename(columns = {0:'LifeExp'},inplace=True)
data_long_with_stack
```

	continent	country	year	LifeExp
0	Africa	Algeria	1952	43.077
1	Africa	Algeria	1957	45.685
2	Africa	Algeria	1962	48.303
3	Africa	Algeria	1967	51.407
4	Africa	Algeria	1972	54.518
...	...	...	...	...
1699	Oceania	New Zealand	1987	74.320
1700	Oceania	New Zealand	1992	76.330
1701	Oceania	New Zealand	1997	77.550
1702	Oceania	New Zealand	2002	79.110
1703	Oceania	New Zealand	2007	80.204

9 Data aggregation

Data aggregation refers to summarizing data with statistics such as sum, count, average, maximum, minimum, etc. to provide a high level view of the data. Often there are mutually exclusive groups in the data that are of interest. In such cases, we may be interested in finding these statistics separately for each group. The Pandas DataFrame method `groupby()` is used to split the data into groups, and then the desired function(s) are applied on each of these groups for groupwise data aggregation. However, the `groupby()` method is not limited to groupwise data aggregation, but can also be used for several other kinds of groupwise operations.

Groupby mechanics: *(Source: https://pandas.pydata.org/docs/user_guide/groupby.html)*

Group by: split-apply-combine By *'group by'* we are referring to a process involving one or more of the following steps:

1. Splitting the data into groups based on some criteria.
2. Applying a function to each group independently.
3. Combining the results in a DataFrame.

Out of these, the split step is the most straightforward. In fact, in many situations we may wish to split the data set into groups and do something with those groups. In the apply step, we may wish to do one of the following:

1. Aggregation: compute a summary statistic (or statistics) for each group. Some examples:

- Compute group sums or means.
- Compute group sizes / counts.

2. Transformation: perform some group-specific computations and return a like-indexed object. Some examples:

- Standardize data *(zscore)* within a group.
- Filling *NAs* within groups with a value derived from each group.

3. Filtration: discard some groups, according to a group-wise computation that evaluates True or False. Some examples:

- Discard data that belongs to groups with only a few members.
- Filter out data based on the group *sum* or *mean*.

Some combination of the above: *GroupBy* will examine the results of the *apply* step and try to return a sensibly combined result if it doesn't fit into either of the above two categories.

We'll use Pandas to group the data and perform *GroupBy* operations.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

9.1 The GroupBy object

9.1.1 Creating a GroupBy object: `groupby`

This Pandas DataFrame method `groupby()` is used to create a `GroupBy` object.

A string passed to `groupby()` may refer to either a column or an index level. If a string matches both a column name and an index level name, a ValueError will be raised.

Example: Consider the life expectancy dataset, *gdp_lifeExpectancy.csv*. Suppose we want to group by the observations by `continent`.

```
gdp_lifeExp_data = pd.read_csv('./Datasets/gdp_lifeExpectancy.csv')
gdp_lifeExp_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

We will pass the column `continent` as an argument to the `groupby()` method.

```
#Creating a GroupBy object
grouped = gdp_lifeExp_data.groupby('continent')
#This will split the data into groups that correspond to values of the column 'continent'
```

The `groupby()` method returns a *GroupBy* object.

```
#A 'GroupBy' objects is created with the `groupby()` function
type(grouped)
```

```
pandas.core.groupby.generic.DataFrameGroupBy
```

The GroupBy object `grouped` contains the information of the groups in which the data is distributed. Each observation has been assigned to a specific group of the column(s) used to group the data. However, note that the dataset is not physically split into different DataFrames. For example, in the above case, each observation is assigned to a particular group depending on the value of the `continent` for that observation. However, all the observations are still in the same DataFrame `data`.

9.1.2 Attributes and methods of the *GroupBy* object

9.1.2.1 `keys`

The object(s) grouping the data are called *key(s)*. Here `continent` is the group key. The keys of the *GroupBy* object can be seen using Its `keys` attribute.

```
#Key(s) of the GroupBy object
grouped.keys
```

```
'continent'
```

9.1.2.2 `ngroups`

The number of groups in which the data is distributed based on the keys can be seen with the `ngroups` attribute.

```
#The number of groups based on the key(s)
grouped.ngroups
```

```
5
```

9.1.2.3 `groups`

The `groups` attribute of the *GroupBy* object contains the group labels (or names) and the row labels of the observations in each group, as a dictionary.

```
#The groups (in the dictionary format)
grouped.groups
```

```
{'Africa': [24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 4
```

The group names are the *keys* of the dictionary, while the row labels are the corresponding *values*

```
#Group names
grouped.groups.keys()
```

```
dict_keys(['Africa', 'Americas', 'Asia', 'Europe', 'Oceania'])
```

```
#Group values are the row labels corresponding to a particular group
grouped.groups.values()
```

```
dict_values([Int64Index([  24,   25,   26,   27,   28,   29,   30,   31,   32,   33,
             ...
             1694, 1695, 1696, 1697, 1698, 1699, 1700, 1701, 1702, 1703],
           dtype='int64', length=624), Int64Index([  48,   49,   50,   51,   52,   53,   54,
             ...
             1634, 1635, 1636, 1637, 1638, 1639, 1640, 1641, 1642, 1643],
           dtype='int64', length=300), Int64Index([   0,    1,    2,    3,    4,    5,    6,
             ...
             1670, 1671, 1672, 1673, 1674, 1675, 1676, 1677, 1678, 1679],
           dtype='int64', length=396), Int64Index([  12,   13,   14,   15,   16,   17,   18,
             ...
             1598, 1599, 1600, 1601, 1602, 1603, 1604, 1605, 1606, 1607],
           dtype='int64', length=360), Int64Index([  60,   61,   62,   63,   64,   65,   66,
               71, 1092, 1093, 1094, 1095, 1096, 1097, 1098, 1099, 1100, 1101,
             1102, 1103],
           dtype='int64')])
```

9.1.2.4 size()

The `size()` method of the *GroupBy* object returns the number of observations in each group.

```
#Number of observations in each group
grouped.size()
```

```
continent
Africa      624
Americas    300
Asia        396
Europe      360
Oceania      24
dtype: int64
```

9.1.2.5 first()

The first non missing element of each group is returned with the `first()` method of the *GroupBy* object.

```
#The first element of the group can be printed using the first() method
grouped.first()
```

continent	country	year	lifeExp	pop	gdpPercap
Africa	Algeria	1952	43.077	9279525	2449.008185
Americas	Argentina	1952	62.485	17876956	5911.315053
Asia	Afghanistan	1952	28.801	8425333	779.445314
Europe	Albania	1952	55.230	1282697	1601.056136
Oceania	Australia	1952	69.120	8691212	10039.595640

9.1.2.6 get_group()

This method returns the observations for a particular group of the *GroupBy* object.

```
#Observations for individual groups can be obtained using the get_group() function
grouped.get_group('Asia')
```

	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
...	...	...	...	...	...	...
1675	Yemen, Rep.	Asia	1987	52.922	11219340	1971.741538
1676	Yemen, Rep.	Asia	1992	55.599	13367997	1879.496673
1677	Yemen, Rep.	Asia	1997	58.020	15826497	2117.484526
1678	Yemen, Rep.	Asia	2002	60.308	18701257	2234.820827
1679	Yemen, Rep.	Asia	2007	62.698	22211743	2280.769906

9.2 Data aggregation with `groupby()` methods

9.2.1 `mean()`

This method returns the mean of each group of the *GroupBy* object.

9.2.1.1 Grouping observations

Example: Find the mean life expectancy, population and GDP per capita for each country since 1952.

First, we'll group the data such that all observations corresponding to a country make a unique group.

```
#Grouping the observations by 'country'
grouped = gdp_lifeExp_data.groupby('country')
```

Now, we'll find the mean statistics for each group with the `mean()` method. The method will be applied on all columns of the DataFrame and all groups.

```
#Finding the mean stastistic of all columns of the DataFrame and all groups
grouped.mean()
```

	year	lifeExp	pop	gdpPercap
country				
Afghanistan	1979.5	37.478833	1.582372e+07	802.674598

country	year	lifeExp	pop	gdpPercap
Albania	1979.5	68.432917	2.580249e+06	3255.366633
Algeria	1979.5	59.030167	1.987541e+07	4426.025973
Angola	1979.5	37.883500	7.309390e+06	3607.100529
Argentina	1979.5	69.060417	2.860224e+07	8955.553783
...	...	...	...	...
Vietnam	1979.5	57.479500	5.456857e+07	1017.712615
West Bank and Gaza	1979.5	60.328667	1.848606e+06	3759.996781
Yemen, Rep.	1979.5	46.780417	1.084319e+07	1569.274672
Zambia	1979.5	45.996333	6.353805e+06	1358.199409
Zimbabwe	1979.5	52.663167	7.641966e+06	635.858042

Note that if we wished to retain the `continent` in the above dataset, we can group the data by both `continent` and `country`. If the data is to be grouped by multiple columns, we need to put them within `[]` brackets:

```
#Grouping the observations by 'continent' and 'country'
grouped = gdp_lifeExp_data.groupby(['continent','country'])

#Finding the mean stastistic of all columns of the DataFrame and all groups
grouped.mean()
```

continent	country	year	lifeExp	pop	gdpPercap
Africa	Algeria	1979.5	59.030167	1.987541e+07	4426.025973
	Angola	1979.5	37.883500	7.309390e+06	3607.100529
	Benin	1979.5	48.779917	4.017497e+06	1155.395107
	Botswana	1979.5	54.597500	9.711862e+05	5031.503557
	Burkina Faso	1979.5	44.694000	7.548677e+06	843.990665
...	...	...	...	...	...
Europe	Switzerland	1979.5	75.565083	6.384293e+06	27074.334405
	Turkey	1979.5	59.696417	4.590901e+07	4469.453380
	United Kingdom	1979.5	73.922583	5.608780e+07	19380.472986
Oceania	Australia	1979.5	74.662917	1.464931e+07	19980.595634
	New Zealand	1979.5	73.989500	3.100032e+06	17262.622813

Here the data has been aggregated according to the group keys - `continent` and `country`, and a new DataFrame is created that is indexed by the unique values of `continent-country`.

For large datasets, it may be desirable to aggregate only a few columns. For example, if we wish to compute the means of only `lifeExp` and `gdpPercap`, then we can filter those columns in the *GroupBy* object *(just like we filter columns in a DataFrame)*, and then apply the `mean()` method:

```
grouped[['lifeExp','gdpPercap']].mean()
```

continent	country	lifeExp	gdpPercap
Africa	Algeria	59.030167	4426.025973
	Angola	37.883500	3607.100529
	Benin	48.779917	1155.395107
	Botswana	54.597500	5031.503557
	Burkina Faso	44.694000	843.990665
...	...	...	...
Europe	Switzerland	75.565083	27074.334405
	Turkey	59.696417	4469.453380
	United Kingdom	73.922583	19380.472986
Oceania	Australia	74.662917	19980.595634
	New Zealand	73.989500	17262.622813

9.2.1.2 Grouping columns

By default, the grouping takes place by rows. However, as with several other Pandas methods, grouping can also be done by columns by using the `axis = 1` argument.

Example: Consider we have the above dataset in the wide-format as follows.

```
gdp_lifeExp_data_wide = gdp_lifeExp_data.pivot(index = ['continent','country'],columns = '
gdp_lifeExp_data_wide
```

		lifeExp								
	year	1952	1957	1962	1967	1972	1977	1982	1987	1992
continent	country									
Africa	Algeria	43.077	45.685	48.303	51.407	54.518	58.014	61.368	65.799	67.744
	Angola	30.015	31.999	34.000	35.985	37.928	39.483	39.942	39.906	40.647
	Benin	38.223	40.358	42.618	44.885	47.014	49.190	50.904	52.337	53.919
	Botswana	47.622	49.618	51.520	53.298	56.024	59.319	61.484	63.622	62.745
	Burkina Faso	31.975	34.906	37.814	40.697	43.591	46.137	48.122	49.557	50.260
...	...	...	...	...	...	...	...	...	...	...

		lifeExp								
	year	1952	1957	1962	1967	1972	1977	1982	1987	1992
continent	country									
Europe	Switzerland	69.620	70.560	71.320	72.770	73.780	75.390	76.210	77.410	78.030
	Turkey	43.585	48.079	52.098	54.336	57.005	59.507	61.036	63.108	66.146
	United Kingdom	69.180	70.420	70.760	71.360	72.010	72.760	74.040	75.007	76.420
Oceania	Australia	69.120	70.330	70.930	71.100	71.930	73.490	74.740	76.320	77.560
	New Zealand	69.390	70.260	71.240	71.520	71.890	72.220	73.840	74.320	76.330

Now, find the mean GDP per capita, life expectancy and population for each country.

Here, we can group by the outer level column labels to obtain the means. Also, we need to use the argument `axis=1` to indicate that we intend to group columns, instead of rows.

```
gdp_lifeExp_data_wide.groupby(axis=1,level=0).mean()
```

		gdpPercap	lifeExp	pop
continent	country			
Africa	Algeria	4426.025973	59.030167	1.987541e+07
	Angola	3607.100529	37.883500	7.309390e+06
	Benin	1155.395107	48.779917	4.017497e+06
	Botswana	5031.503557	54.597500	9.711862e+05
	Burkina Faso	843.990665	44.694000	7.548677e+06
...	...	...	...	...
Europe	Switzerland	27074.334405	75.565083	6.384293e+06
	Turkey	4469.453380	59.696417	4.590901e+07
	United Kingdom	19380.472986	73.922583	5.608780e+07
Oceania	Australia	19980.595634	74.662917	1.464931e+07
	New Zealand	17262.622813	73.989500	3.100032e+06

9.2.2 Practice exercise 1

Read the table consisting of GDP per capita of countries from the webpage: https://en.wikipedia.org/wiki/List_c
.

To only read the relevant table, read the tables that contain the word *'Country'*.

Estimate the GDP per capita of each country as the average of the estimates of the three agencies - IMF, United Nations and World Bank.

We need to do a bit of data cleaning before we could directly use the `groupby()` function. Follow the steps below to answer this question:

1. Set the first 2 columns containing country, and UN region as hierarchical row labels.
2. Apply the following function on all the columns to convert them to numeric: `f = lambda x:pd.to_numeric(x,errors = 'coerce')`
3. Now use `groupby()` to find estimate the GDP per capita for each country.

Solution:

```
dfs = pd.read_html('https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)_per_c
gdp_per_capita = dfs[0]
gdp_per_capita_reindexed = gdp_per_capita.set_index([('Country/Territory','Country/Territo
                                                     ('UN Region','UN Region')])
gdp_per_capita_numeric=gdp_per_capita_reindexed.apply(lambda x:pd.to_numeric(x,errors = 'c
gdp_per_capita_numeric.groupby(axis=1,level=1).mean().drop(columns='Year')
```

9.2.3 agg()

Directly applying the aggregate methods of the *GroupBy* object such as mean, count, etc., lets us apply only one function at a time. Also, we may wish to apply an aggregate function of our own, which is not there in the set of methods of the *GroupBy* object, such as the range of values of a column.

The `agg()` function of a *GroupBy* object lets us aggregate data using:

1. Multiple aggregation functions
2. Custom aggregate functions (in addition to in-built functions like mean, std, count etc.)

Consider the spotify dataset containing information about tracks and artists.

```
spotify_data = pd.read_csv('./Datasets/spotify_data.csv')
spotify_data.head(3)
```

	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

Suppose, we wish to find the average popularity of tracks for each genre. We can do that by using the `mean()` method of a *GroupBy* object, as shown in the previous section. We will also sort the table by decreasing average popularity.

```
grouped=spotify_data.groupby('genres')
grouped['track_popularity'].mean().sort_values(ascending = False)
```

```
genres
rap              51.162959
hip hop          48.700665
metal            46.334539
electronic       43.253165
country          41.132686
pop              37.783194
latin            37.563765
rock             36.749623
miscellaneous    36.167401
pop & rock       35.619242
hoerspiel        31.258670
folk             29.716767
jazz             20.349472
Name: track_popularity, dtype: float64
```

Let us also find the standard deviation of the popularity of the tracks for each genre. We will also sort the table by decreasing standard deviation of popularity.

```
grouped['track_popularity'].std().sort_values(ascending = False)
```

```
genres
rap              20.912240
country          20.544338
pop              17.790385
miscellaneous    16.240129
electronic       16.075841
pop & rock       15.560975
folk             15.150717
jazz             15.087119
latin            14.492199
rock             14.350580
metal            14.331154
hip hop          12.824901
```

```
hoerspiel          6.459370
Name: track_popularity, dtype: float64
```

Even though rap is the most popular genre on an average, its popularity varies the most amongs listeners. So, it should probably be recommended only to rap listeners or the criteria to accept rap songs on Spotify should be more stringent.

9.2.3.1 Multiple aggregate functions

Let us use the `agg()` method of the *GroupBy* object to simultaneously find the mean and standard deviation of the track popularity for each genre.

For aggregating by multiple functions, we pass a list of strings to *agg()*, where the strings are the function names.

```
grouped['track_popularity'].agg(['mean','std']).sort_values(by = 'mean',ascending = False)
```

genres	mean	std
rap	51.162959	20.912240
hip hop	48.700665	12.824901
metal	46.334539	14.331154
electronic	43.253165	16.075841
country	41.132686	20.544338
pop	37.783194	17.790385
latin	37.563765	14.492199
rock	36.749623	14.350580
miscellaneous	36.167401	16.240129
pop & rock	35.619242	15.560975
hoerspiel	31.258670	6.459370
folk	29.716767	15.150717
jazz	20.349472	15.087119

From the above table, we observe that people not just like hip-hop the second most on average, but they also like it more consistently than almost all other genres. We have also sorted the above table by decreasing average track popularity.

9.2.3.2 Custom aggregate functions

In addition to the mean and standard deviation of the track popularirty of each genre, let us also include the 90^{th} percentile of track popularity in the table above, and sort it by the same.

```
#Defining a function that returns the 90th percentile value
def Ninety_pc(x):
    return x.quantile(0.9)
```

```
grouped['track_popularity'].agg(['mean','std',Ninety_pc]).sort_values(by = 'Ninety_pc',asc
```

	mean	std	Ninety_pc
genres			
rap	51.162959	20.912240	74
country	41.132686	20.544338	67
hip hop	48.700665	12.824901	64
metal	46.334539	14.331154	63
electronic	43.253165	16.075841	61
pop	37.783194	17.790385	60
miscellaneous	36.167401	16.240129	57
latin	37.563765	14.492199	56
pop & rock	35.619242	15.560975	56
rock	36.749623	14.350580	56
folk	29.716767	15.150717	50
jazz	20.349472	15.087119	43
hoerspiel	31.258670	6.459370	37

From the above table, we observe that even though country songs are not as popular as hip-hop on an average, the top 10% country tracks are more popular than the top 10% hip hop tracks.

For aggregating by multiple functions & changing the column names resulting from those functions, we pass a list of tuples to `agg()`, where each tuple is of length two, and contains the new column name & the function to be applied.

```
#Simultaneous renaming of columns while grouping
grouped['track_popularity'].agg([('Average','mean'),('Standard deviation','std'),('90th pe
```

genres	Average	Standard deviation	90th percentile
rap	51.162959	20.912240	74
country	41.132686	20.544338	67
hip hop	48.700665	12.824901	64
metal	46.334539	14.331154	63
electronic	43.253165	16.075841	61
pop	37.783194	17.790385	60
miscellaneous	36.167401	16.240129	57
latin	37.563765	14.492199	56
pop & rock	35.619242	15.560975	56
rock	36.749623	14.350580	56
folk	29.716767	15.150717	50
jazz	20.349472	15.087119	43
hoerspiel	31.258670	6.459370	37

We can put use a lambda function as well instead of separately defining the function `Ninety_pc` in the above code:

```
#Simultaneous renaming of columns while grouping
grouped['track_popularity'].agg([('Average','mean'),('Standard deviation','std'),('90th pe
```

genres	Average	Standard deviation	90th percentile
rap	51.162959	20.912240	74
country	41.132686	20.544338	67
hip hop	48.700665	12.824901	64
metal	46.334539	14.331154	63
electronic	43.253165	16.075841	61
pop	37.783194	17.790385	60
miscellaneous	36.167401	16.240129	57
latin	37.563765	14.492199	56
pop & rock	35.619242	15.560975	56
rock	36.749623	14.350580	56
folk	29.716767	15.150717	50
jazz	20.349472	15.087119	43
hoerspiel	31.258670	6.459370	37

9.2.3.3 Grouping by multiple columns

Let us find aggregate statistics when we group data by multiple columns. For that, let us create a new categorical column `energy_lvl` that has two levels - *Low energy* and *High energy*, depending on the `energy` of the track.

```
#Creating a new categorical column 'energy_lvl'
spotify_data['energy_lvl'] = pd.qcut(spotify_data.energy,2,labels=['Low energy', 'High ene
```

Now, let us find the mean, standard deviation and 90th percentile value of the track popularity for each genre-energy level combination simultaneously.

```
#Grouping the data with 'genres' and 'energy_lvl'
grouped=spotify_data.groupby(['genres','energy_lvl'])
```

```
#Finding aggregate statistics of data grouped with multple columns
grouped['track_popularity'].agg(['mean','std',Ninety_pc])
```

		mean	std	Ninety_pc
genres	energy_lvl			
country	Low energy	34.982069	19.877274	64.0
	High energy	49.859100	18.196092	69.0
electronic	Low energy	43.754789	13.294925	59.0
	High energy	43.005671	17.290356	61.2
folk	Low energy	29.617831	15.360910	51.0
	High energy	29.991957	14.556622	49.0
hip hop	Low energy	50.283669	12.423124	66.0
	High energy	48.012067	12.936656	63.0
hoerspiel	Low energy	31.534779	5.953968	37.0
	High energy	30.514032	7.609088	38.0
jazz	Low energy	19.421085	14.599499	41.0
	High energy	25.715373	16.662754	50.0
latin	Low energy	34.308370	15.126721	53.0
	High energy	39.992605	13.504109	58.0
metal	Low energy	38.612403	16.000151	60.4
	High energy	46.985621	13.993685	64.0
miscellaneous	Low energy	34.157235	16.516910	56.0
	High energy	39.394186	15.241620	59.0
pop	Low energy	34.722631	18.340408	59.0
	High energy	40.597155	16.784034	61.0
pop & rock	Low energy	32.987221	15.594202	54.0

genres	energy_lvl	mean	std	Ninety_pc
	High energy	37.413357	15.280915	57.0
rap	Low energy	57.177966	16.491203	75.0
	High energy	48.225166	22.182320	74.0
rock	Low energy	34.654871	14.387263	54.0
	High energy	38.256199	14.133726	57.0

For most of the genres, there is not much difference between the average popularity of low energy and high energy tracks. However, in case of country tracks people seem to to prefer high energy tracks a lot more as compared to low energy tracks.

9.2.3.4 Multiple aggregate functions on multiple columns

Let us find the mean and standard deviation of track popularity and danceability for each genre and energy level.

```
spotify_popularity_danceability = grouped[['track_popularity','danceability']].agg([('Aver
spotify_popularity_danceability
```

	track_popularity		danceability	
	Average	Standard deviation	Average	Standard deviation
genres				
country	41.132686	20.544338	0.600392	0.119116
electronic	43.253165	16.075841	0.610729	0.190133
folk	29.716767	15.150717	0.525161	0.150081
hip hop	48.700665	12.824901	0.719329	0.128615
hoerspiel	31.258670	6.459370	0.661288	0.078628
jazz	20.349472	15.087119	0.518045	0.149548
latin	37.563765	14.492199	0.607981	0.152661
metal	46.334539	14.331154	0.419201	0.125717
miscellaneous	36.167401	16.240129	0.576601	0.178777
pop	37.783194	17.790385	0.588314	0.153629
pop & rock	35.619242	15.560975	0.545856	0.143857
rap	51.162959	20.912240	0.723586	0.133588
rock	36.749623	14.350580	0.520255	0.154649

We get a couple of insights from the above table:

- High energy songs have higher danceability for most genres as expected. However, for hip-hop, country, rock and hoerspiel, even low-energy songs have comparable danceability.
- Hip hop has the highest danceability as expected. However, high energy rap also has relatively high danceability.

9.2.3.5 Distinct aggregate functions on multiple columns

For aggregating by multiple functions, we pass a list of strings to `agg()`, where the strings are the function names.

For aggregating by multiple functions & changing the column names resulting from those functions, we pass a list of tuples to `agg()`, where each tuple is of length two, and contains the new column name as the first object and the function to be applied as the second object of the tuple.

For aggregating by multiple functions such that a distinct set of functions is applied to each column, we pass a dictionary to `agg()`, where the keys are the column names on which the function is to be applied, and the values are the list of strings that are the function names, or a list of tuples if we also wish to name the aggregated columns.

Example: For each genre and energy level, find the mean and standard deviation of the track popularity, and the minimum and maximum values of loudness.

```
#Specifying arguments to the function as a dictionary if distinct functions are to be appl
grouped.agg({'track_popularity':[('Average','mean'),('Standard deviation','std')],'loudnes
```

		track_popularity		loudness	
		Average	Standard deviation	min	max
genres	energy_lvl				
country	Low energy	34.982069	19.877274	-23.163	-4.145
	High energy	49.859100	18.196092	-16.763	-0.716
electronic	Low energy	43.754789	13.294925	-60.000	-4.936
	High energy	43.005671	17.290356	-19.756	-0.533
folk	Low energy	29.617831	15.360910	-28.715	-0.972
	High energy	29.991957	14.556622	-16.383	0.501
hip hop	Low energy	50.283669	12.423124	-25.947	-1.595
	High energy	48.012067	12.936656	-18.273	0.642
hoerspiel	Low energy	31.534779	5.953968	-29.907	-4.910
	High energy	30.514032	7.609088	-22.046	-2.130
jazz	Low energy	19.421085	14.599499	-60.000	-2.962
	High energy	25.715373	16.662754	-27.563	-1.166
latin	Low energy	34.308370	15.126721	-31.897	-2.331

genres	energy_lvl	track_popularity Average	Standard deviation	loudness min	max
metal	High energy	39.992605	13.504109	-18.357	-1.204
	Low energy	38.612403	16.000151	-32.032	-4.311
miscellaneous	High energy	46.985621	13.993685	-16.244	-1.275
	Low energy	34.157235	16.516910	-44.410	-1.409
pop	High energy	39.394186	15.241620	-37.684	1.634
	Low energy	34.722631	18.340408	-41.182	-0.045
pop & rock	High energy	40.597155	16.784034	-27.575	0.330
	Low energy	32.987221	15.594202	-60.000	-1.823
rap	High energy	37.413357	15.280915	-22.234	1.107
	Low energy	57.177966	16.491203	-23.611	-1.218
rock	High energy	48.225166	22.182320	-15.012	0.457
	Low energy	34.654871	14.387263	-42.488	-1.708
	High energy	38.256199	14.133726	-25.291	3.744

From the above table, we observe that high energy songs are always louder than low energy songs. High energy Rock songs can be very loud.

9.3 `apply()`: Data aggregation, filtering & transformation

With the `apply()` method of the *GroupBy* object, we can perform several operations on groups, other than data aggregation.

9.3.1 Filtering data by group

Example: Find the top 3 most popular tracks of each genre in the spotify dataset.

We'll first define a function that sorts a dataset by decreasing track popularity and returns the top 3 rows. Then, we'll apply this function on each group using the `apply()` method of the *GroupBy* object.

```
#Defining the function that finds the top 3 most popular tracks from the dataset 'x'
def top_stats(x,col='track_popularity',n=3):
    return x.sort_values(by=col,ascending = False)[0:n]
```

```
#Defining the groups in the spotify data
grouped_spotify_data = spotify_data.groupby('genres')
```

Now we'll use the `apply()` method to apply the `top_stats()` function on each group of the object `grouped_spotify_data`.

```
#Top 3 tracks of each genre
top3_tracks = grouped_spotify_data.apply(top_stats)
top3_tracks.head()
```

		artist_followers	genres	artist_name	artist_popularity	track_name	tra
genres							
country	4047	4755356	country	Luke Combs	85	Forever After All	82
	2403	1678738	country	Morgan Wallen	88	Wasted On You	81
	16244	1678738	country	Morgan Wallen	88	Whiskey Glasses	81
electronic	83058	7650304	electronic	Daft Punk	86	One More Time	81
	13068	28026432	electronic	Alan Walker	85	Faded	80

The `top_stats()` function is applied to each group, and the results are concatenated internally with the `concat()` function. The output therefore has a hierarchical index whose outer level indices are the group keys.

We can also use a lambda function instead of separately defining the function `top_tracks()`:

```
#Top 3 tracks of each genre - using lambda function
top3_tracks = grouped_spotify_data.apply(lambda x:x.sort_values(by = 'track_popularity',as
top3_tracks.head()
```

		artist_followers	genres	artist_name	artist_popularity	track_name	tra
genres							
country	4047	4755356	country	Luke Combs	85	Forever After All	82
	2403	1678738	country	Morgan Wallen	88	Wasted On You	81
	16244	1678738	country	Morgan Wallen	88	Whiskey Glasses	81
electronic	83058	7650304	electronic	Daft Punk	86	One More Time	81
	13068	28026432	electronic	Alan Walker	85	Faded	80

We can also pass arguments to the `top_stats()` function with the `apply()` function.

Example: Find the most popular artist from each genre.

```
#Applying the 'top_stats()' function to each group of the data (based on 'genre') to find
#Dropping the inner row label as it is not informative
grouped_spotify_data.apply(top_stats,col='artist_popularity',n=1)['artist_name'].droplevel
```

```
genres
country                Morgan Wallen
electronic                 Daft Punk
folk                        Bon Iver
hip hop                       Eminem
hoerspiel               Die drei ???
jazz               Earth, Wind & Fire
latin                         Maluma
metal                    Linkin Park
miscellaneous              Pop Smoke
pop                    Justin Bieber
pop & rock                  Maroon 5
rap                        Bad Bunny
rock                 Imagine Dragons
Name: artist_name, dtype: object
```

9.3.2 Practice exercise 2

Filter the first 4 columns of the spotify dataset. Drop duplicate observartions in the resulting dataset using the Pandas DataFrame method `drop_duplicates()`. Find the top 3 most popular artists for each genre.

Solution:

```
spotify_data.iloc[:,0:4].drop_duplicates().groupby('genres').apply(lambda x:x.sort_values(
```

9.3.3 Transforming data by group

Recall method 3 for imputing missing values in Chapter 7. The method was to impute missing values based on correlated variables in data.

In the example shown for the method, values of GDP per capita for a few countries were missing. We imputed the missing value of GDP per capita for those countries as the average GDP per capita of the corresponding continent.

We will compare the approach we used with the approach using the `groupby()` & `apply()` methods.

Let us read the datasets and the function that makes a visualization to compare the imputed values with the actual values.

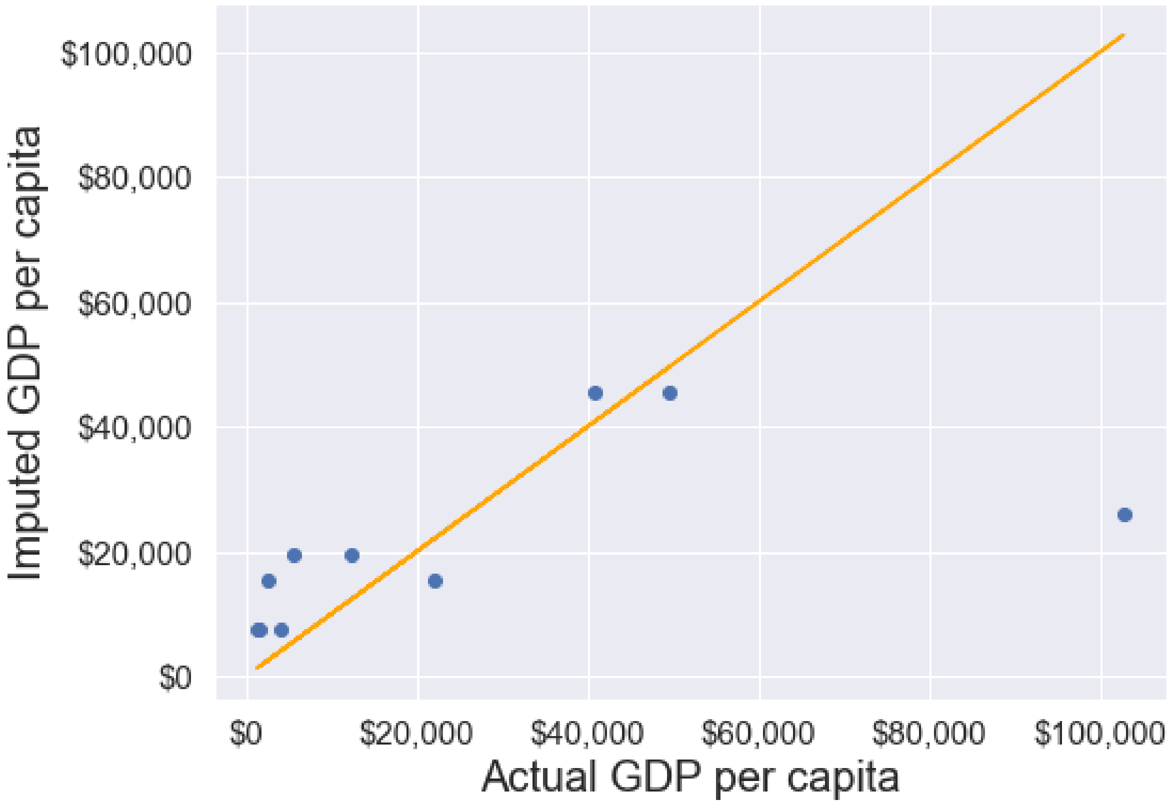
```
#Importing data with missing values
gdp_missing_data = pd.read_csv('./Datasets/GDP_missing_data.csv')

#Importing data with all values
gdp_complete_data = pd.read_csv('./Datasets/GDP_complete_data.csv')
```

```
#Index of rows with missing values for GDP per capita
null_ind_gdpPC = gdp_missing_data.index[gdp_missing_data.gdpPerCapita.isnull()]

#Defining a function to plot the imputed values vs actual values
def plot_actual_vs_predicted():
    fig, ax = plt.subplots(figsize=(8, 6))
    plt.rc('xtick', labelsize=15)
    plt.rc('ytick', labelsize=15)
    x = gdp_complete_data.loc[null_ind_gdpPC,'gdpPerCapita']
    y = gdp_imputed_data.loc[null_ind_gdpPC,'gdpPerCapita']
    plt.scatter(x,y)
    z=np.polyfit(x,y,1)
    p=np.poly1d(z)
    plt.plot(x,x,color='orange')
    plt.xlabel('Actual GDP per capita',fontsize=20)
    plt.ylabel('Imputed GDP per capita',fontsize=20)
    ax.xaxis.set_major_formatter('${x:,.0f}')
    ax.yaxis.set_major_formatter('${x:,.0f}')
    rmse = np.sqrt(((x-y).pow(2)).mean())
    print("RMSE=",rmse)
```

Approach 1: Using the approach we used in Section 7.1.5.3

```
#Finding the mean GDP per capita of the continent
avg_gdpPerCapita = gdp_missing_data['gdpPerCapita'].groupby(gdp_missing_data['continent'])

#Creating a copy of missing data to impute missing values
gdp_imputed_data = gdp_missing_data.copy()

#Replacing missing GDP per capita with the mean GDP per capita for the corresponding conti
for cont in avg_gdpPerCapita.index:
    gdp_imputed_data.loc[(gdp_imputed_data.continent==cont) & (gdp_imputed_data.gdpPerCapi
                     'gdpPerCapita']=avg_gdpPerCapita[cont]
plot_actual_vs_predicted()
```

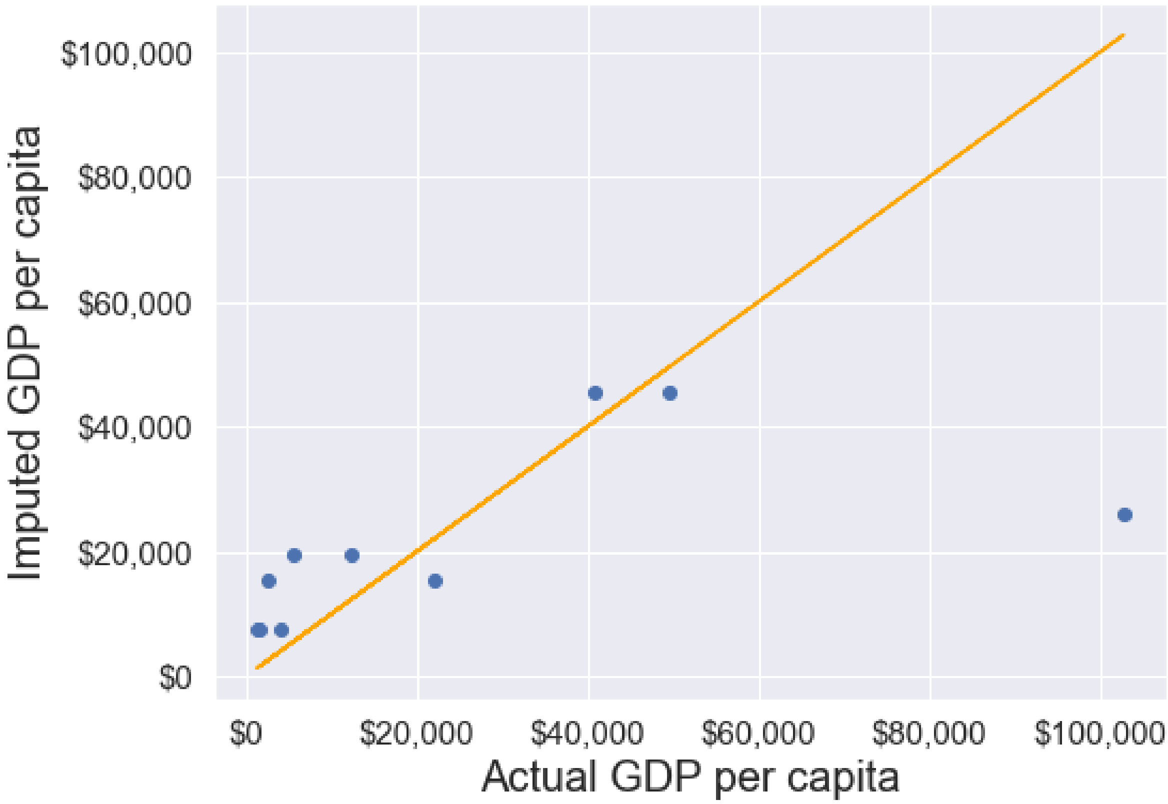
```
RMSE= 25473.20645170116
```



Approach 2: Using the `groupby()` and `apply()` methods.

```
#Creating a copy of missing data to impute missing values
gdp_imputed_data = gdp_missing_data.copy()

#Grouping data by continent
grouped = gdp_missing_data.groupby('continent')

#Applying the lambda function on the 'gdpPerCapita' column of the groups
gdp_imputed_data['gdpPerCapita'] = grouped['gdpPerCapita'].apply(lambda x: x.fillna(x.mean

plot_actual_vs_predicted()
```

```
RMSE= 25473.20645170116
```



With the `apply()` function, the missing value of `gdpPerCapita` for observations of each group are filled by the mean `gdpPerCapita` of that group. The code is not only more convenient to write, but also faster as compared to `for` loops. The `for` loop imputes the missing values of observations of one group at a time, while the imputation may happen in parallel for all groups with the `apply()` function.

9.3.4 Sampling data by group

The `groupby()` and `apply()` method can be used to for stratified random sampling from a large dataset.

The spotify dataset has more than 200k observations. It may be expensive to operate with so many observations. Suppose, we wish to take a random sample of 650 observations to analyze spotify data, such that all genres are equally represented.

Before taking the random sample, let us find the number of tracks in each genre.

```
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
```

Let us take a random sample of 650 observations from the entire dataset.

```
sample_spotify_data = spotify_data.sample(650)
```

Now, let us see the number of track of each genre in our sample.

```
sample_spotify_data.genres.value_counts()
```

```
pop              185
rock             150
miscellaneous    102
pop & rock        98
jazz              37
hoerspiel         25
hip hop           22
rap                7
metal              7
latin              5
country            5
folk               5
electronic         2
Name: genres, dtype: int64
```

Some of the genres have a very low representation in the data. To rectify this, we can take a random sample of 50 observations from each of the 13 genres. In other words, we can take a random sample from each of the genre-based groups.

```
evenly_sampled_spotify_data = spotify_data.groupby('genres').apply(lambda x:x.sample(50))
evenly_sampled_spotify_data.genres.value_counts()
```

```
hoerspiel        50
hip hop          50
pop & rock       50
latin            50
country          50
rap              50
electronic       50
metal            50
folk             50
pop              50
miscellaneous    50
rock             50
jazz             50
Name: genres, dtype: int64
```

The above stratified random sample equally represents all the genres.

9.4 `corr()`: Correlation by group

The `corr()` method of the *GroupBy* object returns the correlation between all pairs of columns within each group.

Example: Find the correlation between danceability and track popularity for each genre-energy level combination.

```
spotify_data.groupby(['genres','energy_lvl']).apply(lambda x:x['danceability'].corr(x['tra
```

```
genres          energy_lvl
country         Low energy    -0.171830
                High energy   -0.154823
electronic      Low energy     0.378330
                High energy    0.072343
folk            Low energy     0.187482
                High energy    0.230419
hip hop         Low energy     0.113421
                High energy    0.027074
```

```
hoerspiel      Low energy    -0.053908
               High energy   -0.044211
jazz           Low energy     0.005733
               High energy    0.332356
latin          Low energy    -0.083971
               High energy    0.030276
metal          Low energy     0.127439
               High energy    0.256165
miscellaneous  Low energy     0.163185
               High energy    0.148818
pop            Low energy     0.208942
               High energy    0.156764
pop & rock     Low energy     0.063127
               High energy    0.060195
rap            Low energy    -0.008394
               High energy   -0.129873
rock           Low energy     0.027876
               High energy    0.065908
dtype: float64
```

The popularity of low energy electronic music is the most correlated with its danceability.

9.5 pivot_table()

The Pandas `pivot_table()` function is used to aggregate data groupwise where some of the group keys are along the rows and some along the columns. Note that `pivot_table()` is the same as `pivot()` except that `pivot_table()` aggregates the data as well in addition to re-arranging it.

Example: Find the mean of track popularity for each genre-energy lvl combination such that each row corresponds to a genre, and the energy levels correspond to columns.

```
pd.pivot_table(data = spotify_data,values = 'track_popularity',index = 'genres', columns =
```

energy_lvl genres	Low energy	High energy	All
country	34.982069	49.859100	41.132686
electronic	43.754789	43.005671	43.253165
folk	29.617831	29.991957	29.716767
hip hop	50.283669	48.012067	48.700665

energy_lvl genres	Low energy	High energy	All
hoerspiel	31.534779	30.514032	31.258670
jazz	19.421085	25.715373	20.349472
latin	34.308370	39.992605	37.563765
metal	38.612403	46.985621	46.334539
miscellaneous	34.157235	39.394186	36.167401
pop	34.722631	40.597155	37.783194
pop & rock	32.987221	37.413357	35.619242
rap	57.177966	48.225166	51.162959
rock	34.654871	38.256199	36.749623
All	33.015545	39.151701	36.080772

We can use also use custom *GroupBy* aggregate functions with `pivot_table()`.

Example: Find the 90^{th} percentile of track popularity for each genre-energy lvl combination such that each row corresponds to a genre, and the energy levels correspond to columns.

```
pd.pivot_table(data = spotify_data,values = 'track_popularity',index = 'genres', columns =
```

energy_lvl genres	Low energy	High energy
country	64.0	69.0
electronic	59.0	61.2
folk	51.0	49.0
hip hop	66.0	63.0
hoerspiel	37.0	38.0
jazz	41.0	50.0
latin	53.0	58.0
metal	60.4	64.0
miscellaneous	56.0	59.0
pop	59.0	61.0
pop & rock	54.0	57.0
rap	75.0	74.0
rock	54.0	57.0

9.6 crosstab()

The `crosstab()` method is a special case of a pivot table for computing group frequncies (or size of each group). We may often use it to check if the data is representative of all groups that are of interest to us.

Example: Find the number of observations in each group, where each groups corresponds to a distinct genre-energy lvl combination

```
#Cross tabulation of 'genres' and 'energy_lvl'
pd.crosstab(spotify_data.genres,spotify_data.energy_lvl,margins = True).sort_values(by = '
```

energy_lvl genres	Low energy	High energy	All
All	121708	121482	243190
pop	33742	36699	70441
rock	20827	28958	49785
pop & rock	17607	25830	43437
miscellaneous	22088	13760	35848
jazz	11392	1971	13363
hoerspiel	9129	3385	12514
hip hop	2235	5138	7373
folk	2075	746	2821
latin	908	1217	2125
rap	590	1208	1798
metal	129	1530	1659
country	725	511	1236
electronic	261	529	790

The above table can be generated with the `pivot_table()` function using *'count'* as the `aggfunc` argument, as shown below. However, the `crosstab()` function is more compact to code.

```
#Generating cross-tabulation of 'genres' and 'energy_lvl' with 'pivot_table()'
pd.pivot_table(data = spotify_data,values = 'track_popularity',index = 'genres', columns =
```

energy_lvl genres	Low energy	High energy	All
country	725	511	1236
electronic	261	529	790

energy_lvl genres	Low energy	High energy	All
folk	2075	746	2821
hip hop	2235	5138	7373
hoerspiel	9129	3385	12514
jazz	11392	1971	13363
latin	908	1217	2125
metal	129	1530	1659
miscellaneous	22088	13760	35848
pop	33742	36699	70441
pop & rock	17607	25830	43437
rap	590	1208	1798
rock	20827	28958	49785
All	121708	121482	243190

Example: Find the percentage of observations in each group of the above table.

```
pd.crosstab(spotify_data.genres,spotify_data.energy_lvl,margins = True).sort_values(by = '
```

energy_lvl genres	Low energy	High energy	All
All	50.046466	49.953534	100.000000
pop	13.874748	15.090670	28.965418
rock	8.564086	11.907562	20.471648
pop & rock	7.240018	10.621325	17.861343
miscellaneous	9.082610	5.658127	14.740738
jazz	4.684403	0.810477	5.494881
hoerspiel	3.753855	1.391916	5.145771
hip hop	0.919034	2.112751	3.031786
folk	0.853242	0.306756	1.159998
latin	0.373371	0.500432	0.873802
rap	0.242609	0.496731	0.739340
metal	0.053045	0.629138	0.682183
country	0.298121	0.210124	0.508245
electronic	0.107323	0.217525	0.324849

9.6.1 Practice exercise 3

What percentage of unique tracks are contributed by the top 5 artists of each genre?

Hint: Find the top 5 artists based on `artist_popularity` for each genre. Count the total number of unique tracks (`track_name`) contributed by these artists. Divide this number by the total number of unique tracks in the data. The `nunique()` function will be useful.

Solution:

```
top5artists = spotify_data.iloc[:,0:4].drop_duplicates().groupby('genres').apply(lambda x:
                                ascending = False)[0:5]).droplevel(axis=0,level=1)['artist_nam
top5artists_tracks = spotify_data.loc[spotify_data.artist_name.isin(top5artists),:]
top5artists_tracks.track_name.nunique()/spotify_data.track_name.nunique()
```

10 Alternative solutions by students

This chapter will consist of solutions suggested by students of the STAT303-1 Fall 2022 class, which are more efficient than those presented in the book. However, these solutions do not replace those in the book to preserve the simplicity of the book.

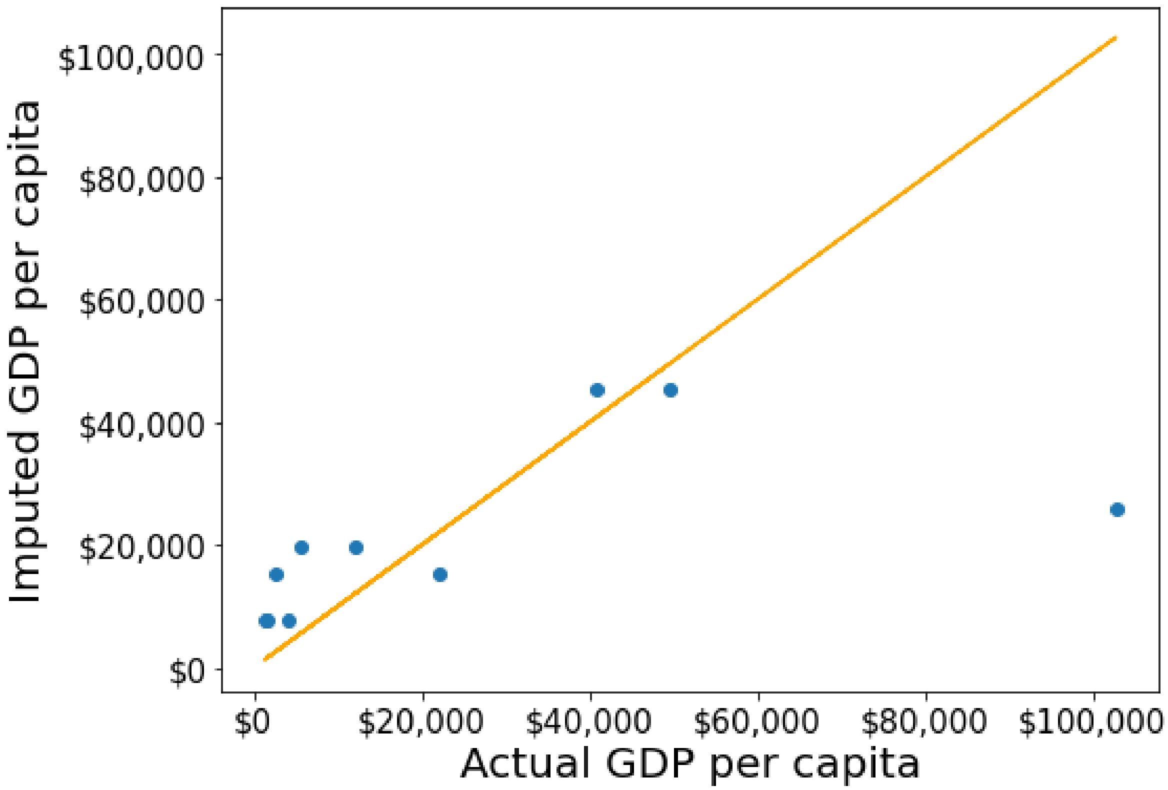
10.1 Missing value imputation based on correlated variables in the data

The code below refers to **Section 7.1.5.3** of the book. Students have proposed some ways to avoid a `for` loop in the code, which will lead to parallel computations, thereby saving execution time.

10.1.1 Original code (in the book)

```
start_time = tm.time()
gdp_imputed_data = gdp_missing_values_data.copy()
for cont in avg_gdpPerCapita.index:
    gdp_imputed_data.loc[(gdp_imputed_data.continent==cont) & (gdp_imputed_data.gdpPerCapi
                     'gdpPerCapita']=avg_gdpPerCapita[cont]
plot_actual_vs_predicted()
print("Time taken to execute code: ",np.round(tm.time()-start_time,3), "seconds")
```

```
RMSE= 25473.20645170116
Time taken to execute code:  0.032 seconds
```



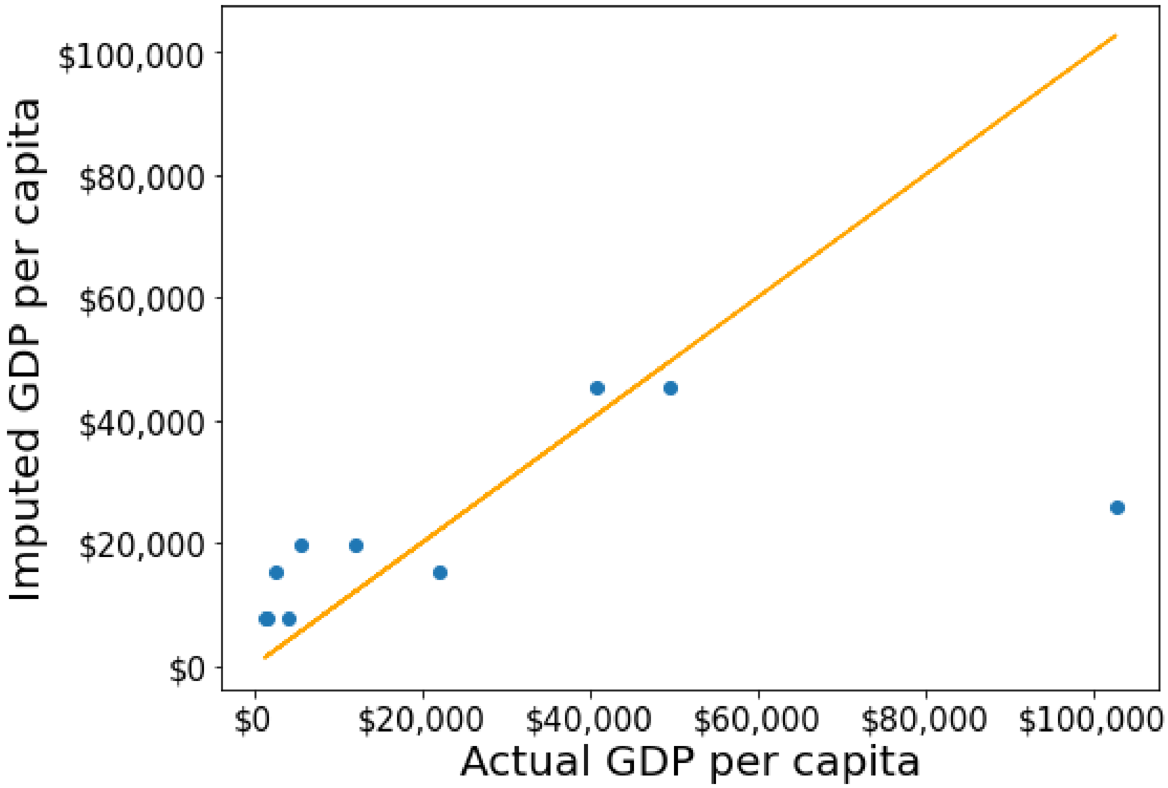
Below are some more efficient ways to impute the missing values, as they avoid using the `for` loop.

10.1.2 Alternative code 1:

By Victoria Shi

```
start_time = tm.time()
gdp_imputed_data = gdp_missing_values_data.copy()
gdp_imputed_data["gdpPerCapita"] =  gdp_missing_values_data[['continent','gdpPerCapita']].
plot_actual_vs_predicted()
print("Time taken to execute code: ",np.round(tm.time()-start_time,3), "seconds")
```

```
RMSE= 25473.20645170116
Time taken to execute code:  0.024 seconds
```

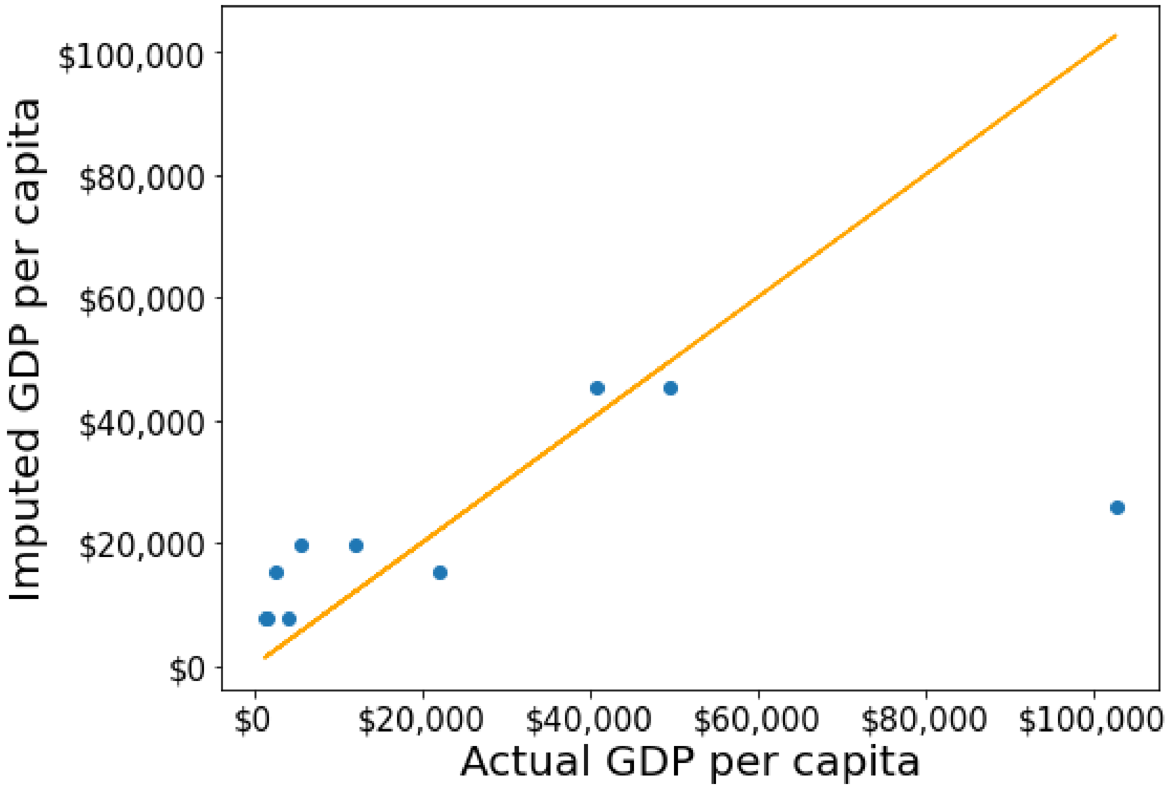


10.1.3 Alternative code 2:

By Elijah Nacar

```
start_time = tm.time()
gdp_imputed_data = gdp_missing_values_data.copy()
gdp_imputed_data.gdpPerCapita = gdp_imputed_data.apply(lambda x: avg_gdpPerCapita[x['conti
plot_actual_vs_predicted()
print("Time taken to execute code: ",np.round(tm.time()-start_time,3), "seconds")
```

```
RMSE= 25473.20645170116
Time taken to execute code:  0.024 seconds
```



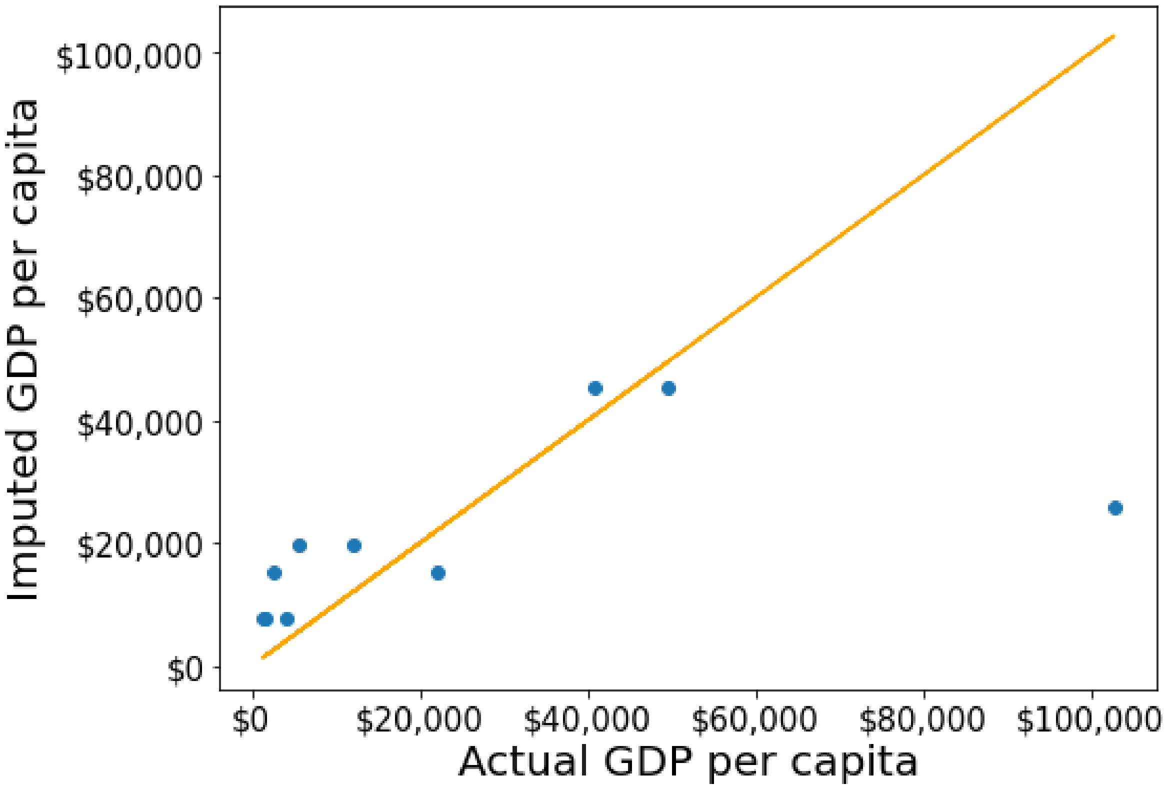
10.1.4 Alternative code 3:

By Erica Zhou

```
start_time = tm.time()
gdp_imputed_data = gdp_missing_values_data.copy()

cond_list = [gdp_imputed_data['continent'] == 'Africa',gdp_imputed_data['continent'] == 'A
             gdp_imputed_data['continent'] == 'Europe',gdp_imputed_data['continent'] == 'N
             gdp_imputed_data['continent'] == 'Oceania',gdp_imputed_data['continent'] == '

choice_list = list(avg_gdpPerCapita)

gdp_imputed_data.gdpPerCapita = np.select(cond_list, choice_list, gdp_imputed_data.gdpPerC
plot_actual_vs_predicted()
print("Time taken to execute code: ",np.round(tm.time()-start_time,3), "seconds")
```

```
RMSE= 25473.20645170116
```

```
Time taken to execute code:  0.024 seconds
```



10.2 Binning with equal sized bins

The code below refers to **Section 7.2.2** of the book. A student has proposed a way to avoid the `for` loop in the code, which will lead to parallel computations, thereby saving execution time.

10.2.1 Original code (in the book)

```
#Bootstrapping to find 95% confidence intervals of Graduation Rate of US universities base
start_time = tm.time()
for expend_bin in college.Expend_bin.unique():
    data_sub = college.loc[college.Expend_bin==expend_bin,:]
    samples = np.random.choice(data_sub['Grad.Rate'], size=(10000,data_sub.shape[0]))
    print("95% Confidence interval of Grad.Rate for "+expend_bin+" univeristies = ["+str(n
```

```
print("Time taken to execute code: ",np.round(tm.time()-start_time,3), "seconds")
```

```
95% Confidence interval of Grad.Rate for Low expend univeristies = [55.31,59.35]
95% Confidence interval of Grad.Rate for High expend univeristies = [71.03,74.9]
95% Confidence interval of Grad.Rate for Med expend univeristies = [64.17,67.95]
Time taken to execute code:  0.151 seconds
```

10.2.2 Alternative code:

By Victoria Shi

```
start_time = tm.time()
def confidence_interval(df):
    samples = np.random.choice(df['Grad.Rate'], size=(10000, df.shape[0]))
    print("95% Confidence interval of Grad.Rate for "+df["Expend_bin"].iloc[0]+" univerist
college.groupby('Expend_bin').apply(confidence_interval)
print("Time taken to execute code: ",np.round(tm.time()-start_time,3), "seconds")
```

```
95% Confidence interval of Grad.Rate for Low expend univeristies = [55.35,59.35]
95% Confidence interval of Grad.Rate for Med expend univeristies = [64.16,67.95]
95% Confidence interval of Grad.Rate for High expend univeristies = [71.05,74.96]
Time taken to execute code:  0.139 seconds
```

A Datasets & Templates

Datasets used in the book, and assignment / project templates can be found here

References

Rubin, Donald B. 1976. "Inference and Missing Data." *Biometrika* 63 (3): 581–92.