DATA ANALYSIS WITH PYTHON

Associate Professor Dr. Nor Badrul Anuar Jumaat

DATA ANALYSIS WITH PANDAS

Module 1 : Introduction to Python & Jupyter Notebooks

Introduction to Python & Jupyter Notebook

What is Python?

- Python is a high-level programming language
- Created by Guido van Rossum and first released in 1991

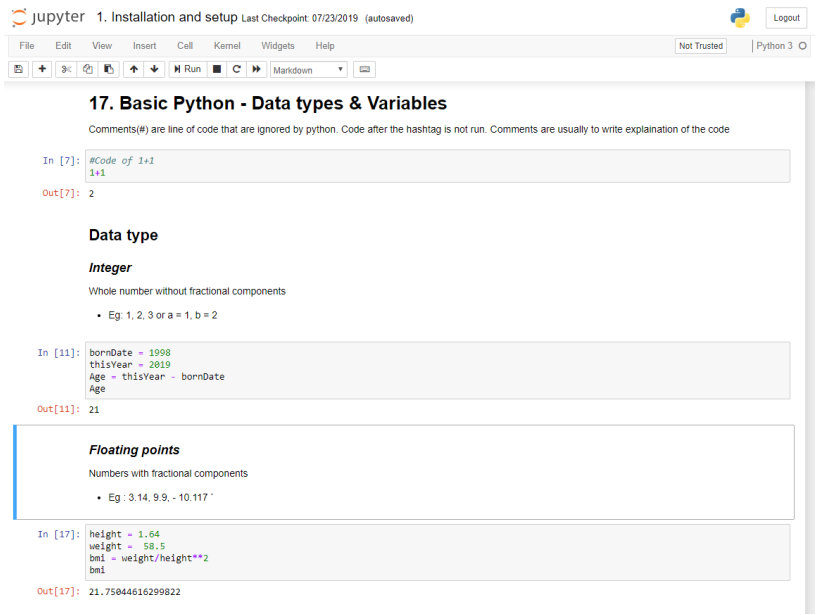
What is Pandas?

- Pandas is a library that is built on top of Python programming language.
- The focus of Pandas is data analysis in which we can analyze, filter, manipulate, and merge data.
- It is almost the same as Spreadsheet or Microsoft Excel
- Pandas is powerful if we need to deal with datasets which have millions of rows

Introduction to Python & Jupyter Notebook

Jupyter Notebook as IDE

- The text editor that we are going to use for this workshop is Jupyter Notebook.
- Working with Jupyter Notebook is much easier as we can instantly see the output below each cell.
- We will get deeper on Jupyter Notebook after this.

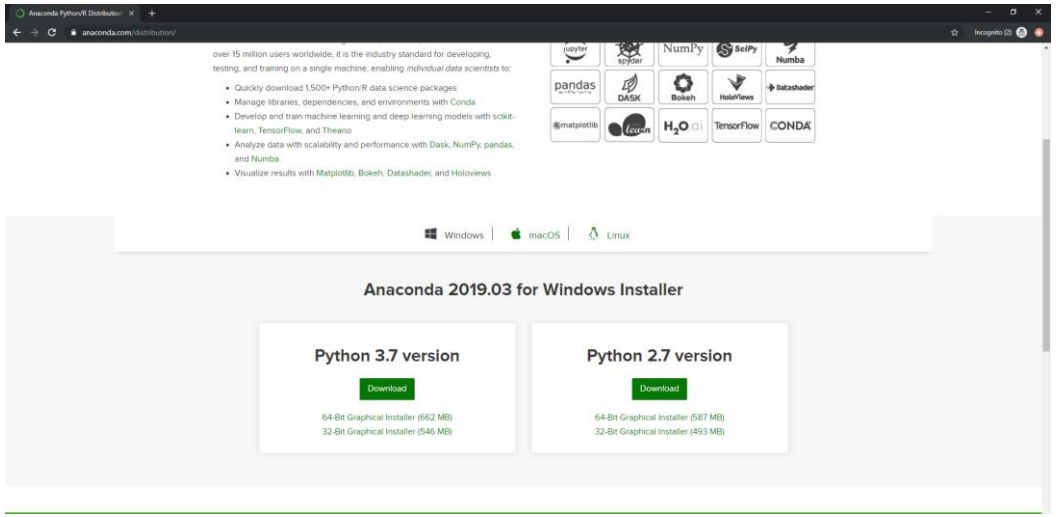


Introduction to Python & Jupyter Notebook

. Installing Anaconda

- The easiest option to start coding with Python is using Anaconda distribution. It comes with Python bundles, pandas and 100+ data analysis libraries.
- Another great thing about Anaconda is that, it is completely **free.**
- To start using it, first, go to Anaconda's official site:
- https://www.anaconda.com/distribution/

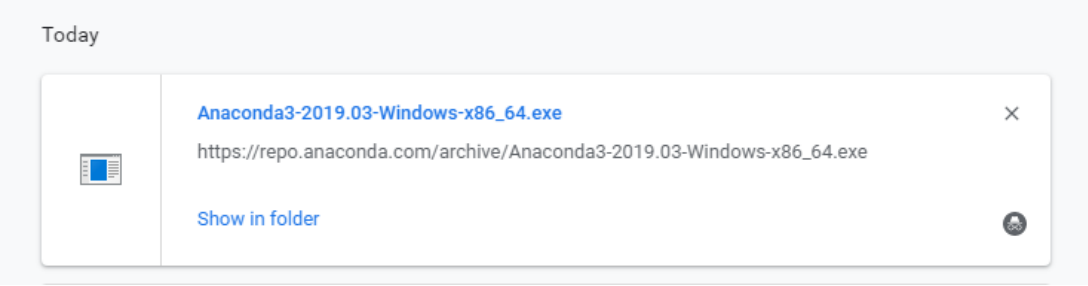
- Click download and you will arrive at the page displayed below.



- Download the installer based on the Operating System that you use.

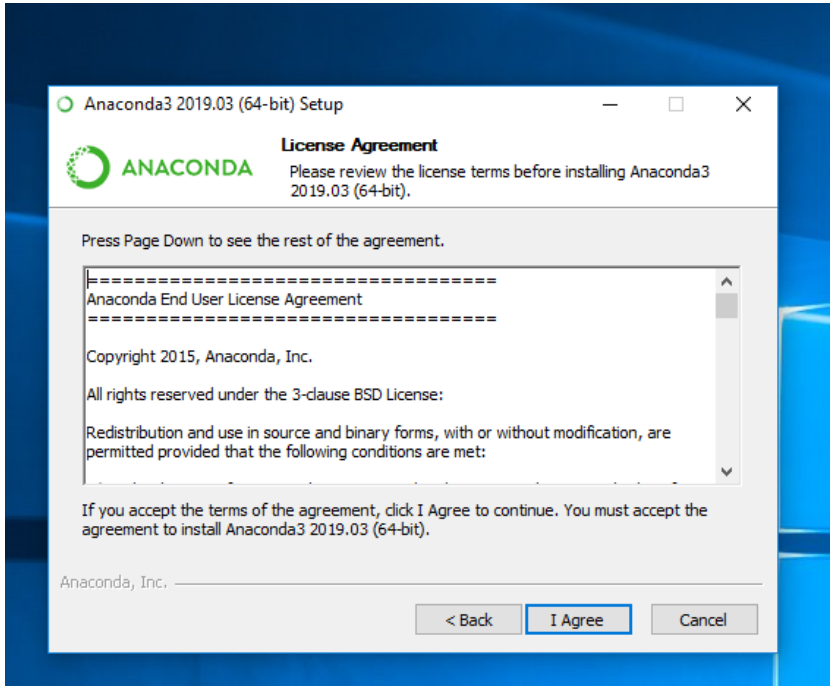
Introduction to Python & Jupyter Notebook

. Installing Anaconda



- Once the download is finished, open the file to install it.

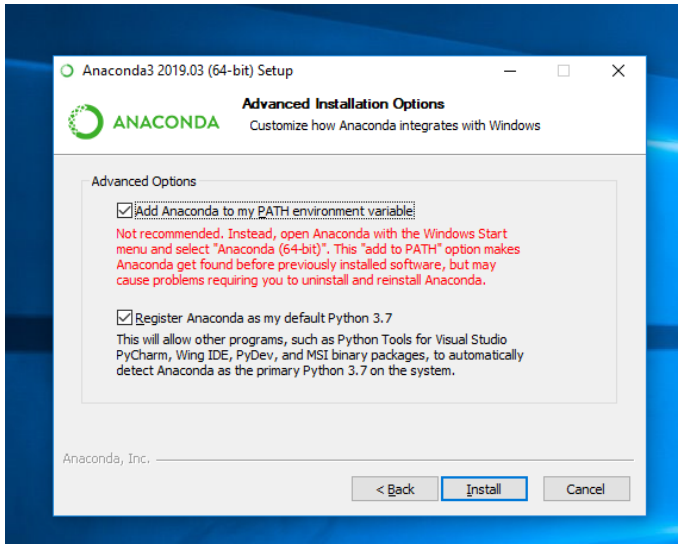
- Click **Next**, and then **I Agree** to start the installation process.



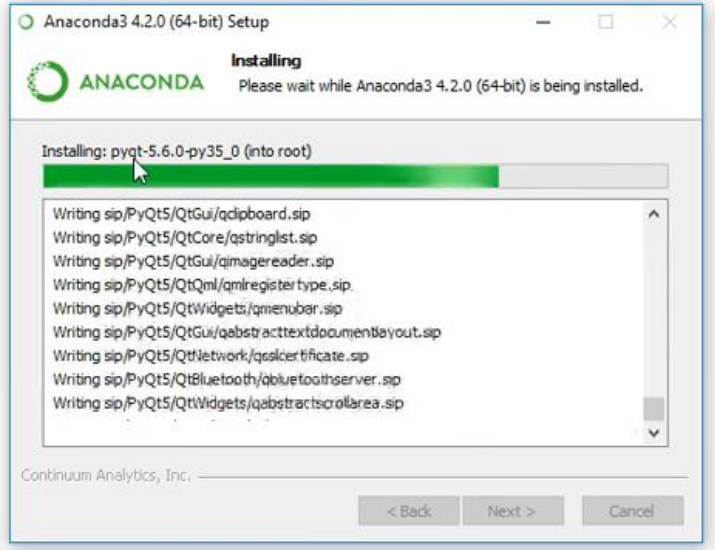
Introduction to Python & Jupyter Notebook

. Installing Anaconda

- For this section, it is optional to tick on the first option.



- Now, wait until the installation is completed.



Introduction to Python & Jupyter Notebook

Update libraries using Command Prompt (CMD)

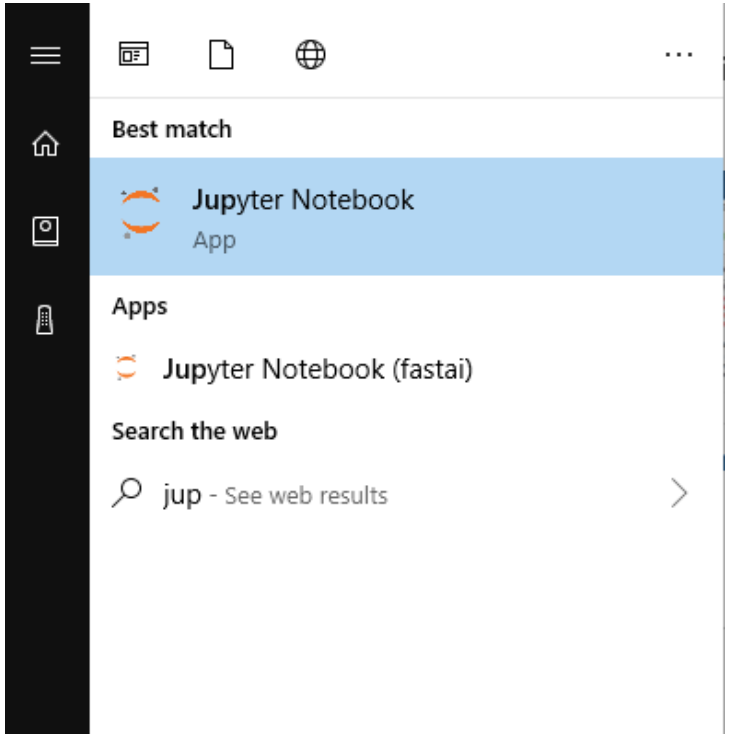
- Click the Windows button and search for CMD
- Type **conda update conda** in the command prompt to update Conda libraries.
- As you can see here, we are trying to update the latest library of **Conda**. The version we have now is 4.5, meanwhile, the latest one is 4.6. It is always better to use the latest library if we can.

```
C:\WINDOWS\system32>conda update conda
Solving environment: done

## Package Plan ##

  environment location: C:\ProgramData\Anaconda3

  added / updated specs:
    - conda


The following packages will be UPDATED:

    conda: 4.5.11-py36_0 --> 4.6.14-py36_0

Proceed ([y]/n)? y

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

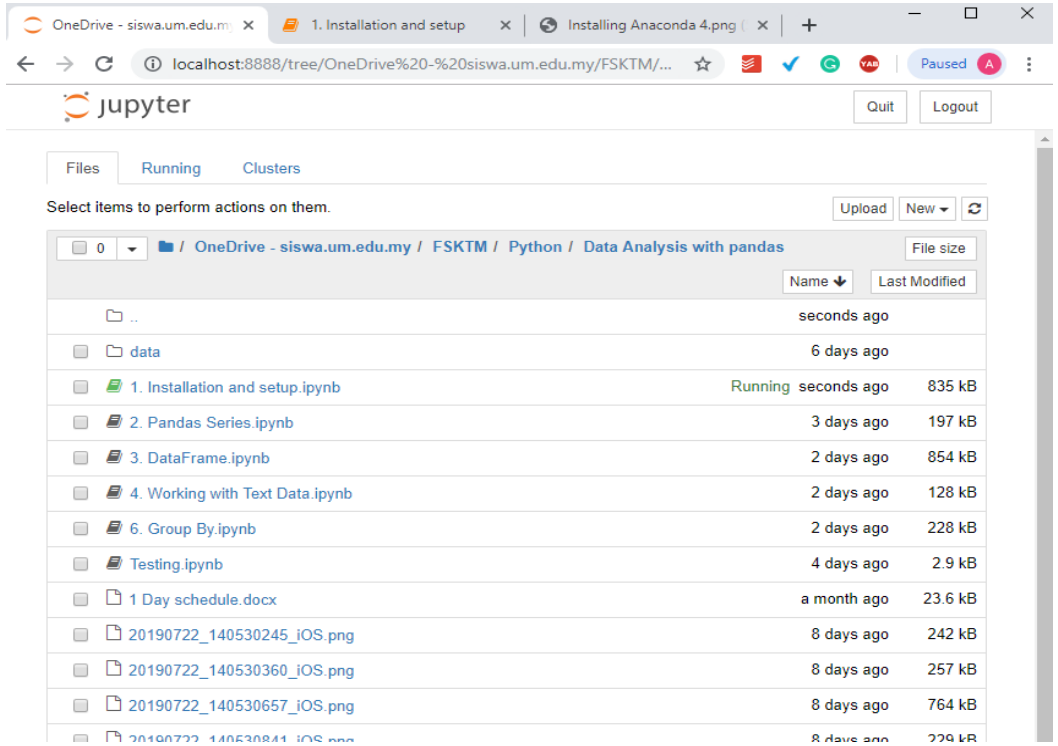
Introduction to Python & Jupyter Notebook

Starting Jupyter Programme

- Search for Jupyter Notebook and run the program.



- You should see the notebook open in your browser.



Introduction to Python & Jupyter Notebook

Import libraries into Jupyter Notebook

- Libraries are prewritten codes that will extend the functionality of python programming language.
- They are managed by Anaconda distribution.
- However, libraries are not imported by default to save memory.
- To import a library you have to type:

import LibraryName **as** aliasName

```
In [2]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

Introduction to Python & Jupyter Notebook

- **import pandas as pd**
  Pandas is a software library written for the Python programming language for data manipulation and analysis. Community standards use **pd as a short form** for pandas.

  import numpy as np
  Numpy stands for Numerical python. It is a library which Pandas is built on top of. We can generate 1 million random rows of random data using it.

  import matplotlib.pyplot as plt
  Rather than importing the whole library, here I choose a specific module .
  This line is for rendering the graph inside the Jupyter Notebook, not in a new window terminal.

Introduction to Python & Jupyter Notebook

Import libraries into Jupyter Notebook

- Test whether the import is working or not.

```
In [4]: pd.__version__
Out[4]: '0.23.4'
```

Introduction to Python & Jupyter Notebook

Basic Python - Data types & Variables

- **Comment(#)** is a line of code that is ignored by python.
- Code after the hashtag is not run.
- Comments are usually used to write the explanation of the code.

```
In [7]: #Code of 1+1
        1+1

Out[7]: 2
```

Introduction to Python & Jupyter Notebook

Data type : Integer & Floating point

Integer

- Whole numbers or numbers without fractional components
- Eg: 1, 2, 3 or a = 1, b = 2

```
In [11]: bornDate = 1998
         thisYear = 2019
         Age = thisYear - bornDate
         Age

Out[11]: 21
```

Floating points

- Numbers with fractional components
- Eg : 3.14, 9.9, - 10.117

```
In [17]: height = 1.64
         weight =  58.5
         bmi = weight/height**2
         bmi

Out[17]: 21.75044616299822
```

Introduction to Python & Jupyter Notebook

Data type : Strings

- **Strings**
- Let's talk about string. A string is a collection of character(s) inside a double quotation mark(""). For example :
- We can also **combine** string with float number using **.format** function.

```
In [18]: "My name is Amin Hakim."

Out[18]: 'My name is Amin Hakim.'
```

```
In [20]: "My height is {}m and my weight is {}kg".format(height, weight)

Out[20]: 'My height is 1.64m and my weight is 58.5kg'
```

Introduction to Python & Jupyter Notebook

Data type : Boolean

- Boolean is a datatype for True or False.
- It is useful when we want to set a condition.
- Boolean can also be used to represent On/Off or Yes/No.
- Bool stands for boolean.

```
In [25]: False

Out[25]: False

In [26]: True

Out[26]: True

In [27]: type(True)

Out[27]: bool
```

Introduction to Python & Jupyter Notebook

Data type

- In python, there is a **built-in function** to let us know what data type we are working with.
- It is called the **type()** function. Inside the bracket, we can specify the data that we want to identify its datatype.
- **int** stands for Integer.
- **str** stands for Strings

```
In [23]: type("I'm eating bread for breakfast")
Out[23]: str

In [24]: type(12345)
Out[24]: int
```

Introduction to Python & Jupyter Notebook

Variables

- Variables are used to store data so that they can be used multiple times. Variables can also store complex mathematical expressions. You can see in the previous example on how we calculate the BMI using height and weights.
- Everything on the right side is calculated first before being assigned to the variable.

```
In [33]: money = 35
         money

Out[33]: 35

In [34]: money = money + 5
         money

Out[34]: 40

In [36]: firstName = "Amin Hakim"
         lastname = "Sazali"

         Full_Name = firstName + " " + lastname
         print(Full_Name)

         Amin Hakim Sazali
```

Introduction to Python & Jupyter Notebook

Basic Python 2 - Lists

- A list is a data container that can store multiple values. In other programming languages, lists are known as arrays.

```
In [37]: [1,2,3,4]
Out[37]: [1, 2, 3, 4]

In [38]: ["John", "Ali", "Abu"]
Out[38]: ['John', 'Ali', 'Abu']
```

- In python, lists can contain **mixed data types**. However, it is more **recommended** to set a **fixed data type** to be used in a list.

```
In [42]: listA = ["John", "Ali", 45, 50, 24.5, 26.7]
         print(listA)

         ['John', 'Ali', 45, 50, 24.5, 26.7]
```

- There is a built-in function in python to calculate the length of the list called **len()**.

```
In [43]: len(listA)
Out[43]: 6
```

Introduction to Python & Jupyter Notebook

Basic Python 2 - Lists

- Every list has its index and it starts with 0.

```
In [45]: name = ["John", "Ali", "Abu", "Aliff"]
         print(name[0])

John
```

- We can also access the elements of the list using **negative number**. Negative numbers **indicates that we count backwards starting from the end of the list.**

```
In [46]: print(name[-1])

Aliff
```

Introduction to Python & Jupyter Notebook

Basic Python 3 - Dictionary

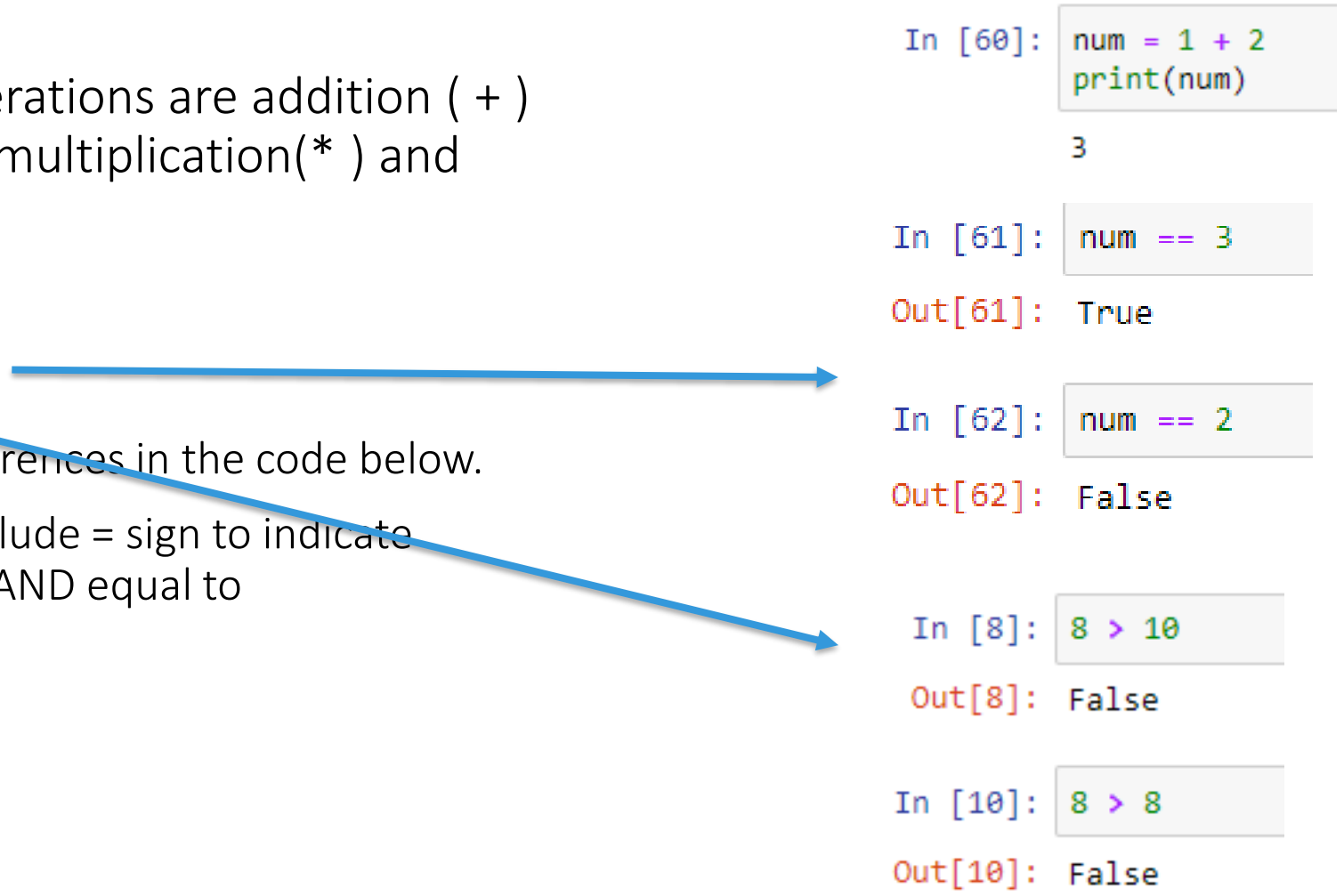
- Python dictionary is one-to-one mapping of **keys and values**. It is just like a traditional dictionary whereby the key is the word and the value is the explanation or description of it .
- In other programming languages, it is known as **map**.
- The **key is unique** and cannot be repeated but multiple keys can have the same value.
- Here is an example of a dictionary for a menu including the prices.

```
In [53]: menu = {
             "Filet-O-Fish" : 10,
             "Starbucks" : 12.50,
             "Fried Rice" : 5.99,
             "Fried Noodle" : 5.99,
             "Big Mac" : 8.75

         }

In [54]: menu["Fried Rice"]

Out[54]: 5.99
```

Introduction to Python & Jupyter Notebook

Basic Python 4 - Operators

- Mathematical operations are addition (+) , subtraction (-), multiplication(*) and division (/).
- Boolean operator
- Range operator
  - Notice the differences in the code below.
  - We can also include = sign to indicate bigger/smaller AND equal to



Introduction to Python & Jupyter Notebook

Basic Python 5 - Functions

- Functions are reusable chunks of code which can be used over and over again.
- They are used to simplify long code and make it more readable.
- A function can have parameter(s) or argument(s) for passing data to itself.

```
In [25]: def convert_to_dollar(ringgit):
             return ringgit / 4.11
```

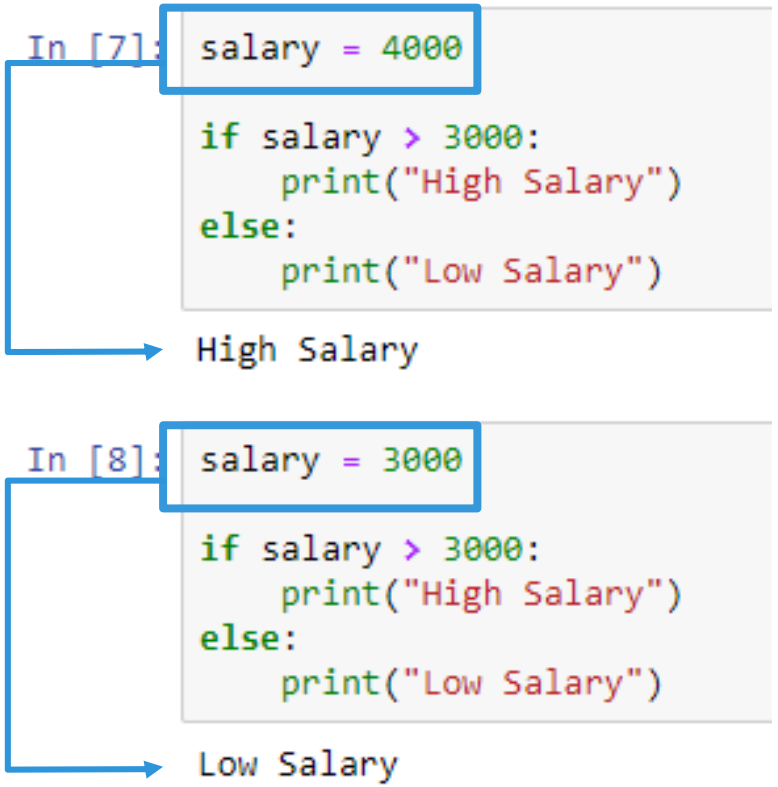
Here we can use the function to convert RM 10 to US Dollar

```
In [26]: convert_to_dollar(10)
Out[26]: 2.4330900243309
```

Introduction to Python & Jupyter Notebook

Basic Python 6 – Conditional Statement

- If we want to have conditions to execute our code, then we would need to use the **IF statement.**
- For instance, when we want to set a threshold for a high salary and a low salary. We define the high salary as more than RM3000 and less than that will be a low salary.
- Changing the salary value to 2000, 5000 and 3000 in the example will give us different outputs.



Introduction to Python & Jupyter Notebook

Basic Python 6 – Conditional Statement

- Notice that, when the salary is equal to 3000, it is considered as Low Salary.
- This is because ”>” notation means **more than**.
- Meanwhile, the notation **”>=“** means **more than or equal to**.

Symbols	Meaning
>	More than
>=	More than or equal to
<	Less than
<=	Less than or equal to

Introduction to Python & Jupyter Notebook

Basic Python 6 – Conditional Statement with List

- If we want to check elements in the list, we can also use the **if** statement combines with **in keyword.**

```
In [15]: items = ["orange", "apple", "mango", "papaya","carrot", "butterhead"]

         if "orange" in items:
             print("Yes! Orange in the list.")
```

```
Yes! Orange in the list.
```

```
In [14]: ic_or_email = "aminhakimsazali@gmail.com"
         # ic_or_email = "570121-87-2911"

         if "@" in ic_or_email:
             print("Email:", ic_or_email )
         else:
             print("IC No.:" , ic_or_email )
```

```
Email: aminhakimsazali@gmail.com
```

Introduction to Python & Jupyter Notebook

Basic Python 6 – Conditional Statement with List

- Two or more conditions? NO PROBLEM!!

```
In [28]: items = ["papaya","carrot", "butterhead"]

         if ("orange" or "apple") in items:
             print("Fruits in the list")
         elif "carrot" in  items:
             print("Vegetable in the list.")
         elif "butterhead" in items:
             print("Butterhead in the list.")
         else:
             print("You dont buy any fruit or vegetables! Eat healthy!")

Vegetable in the list.
```

- Notice that, even though both *carrot* and *butterhead* are in the list, only one statement will be printed. This is because, once an **if else** condition is met, the other conditions after it is completely ignored.

Introduction to Python & Jupyter Notebook

Basic Python 6 – Loops

- There are two loops in Python.
  - For loops
  - While loops
- **For** loops iterate in sequences. The example below will print out the numbers from 0 to the number before 5 (which is 4).

```
In [29]: for i in range(5):
             print(i)
```

```
0
1
2
3
4
```

Introduction to Python & Jupyter Notebook

Basic Python 6 – Loops

- The easiest circumstance is when we want to print six *Hello World*'s. In the first method, we **copy and paste** the same statement 6 times while in the second method, we simply enclosed that statement inside a **for** loop.
- Both methods will give similar result despite the second one having a much shorter code.
- **Code less and smarter!**

```
In [30]: print("Hello World")
         print("Hello World")
         print("Hello World")
         print("Hello World")
         print("Hello World")
         print("Hello World")

Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

```
In [32]: for i in range(6):
             print("Hello World")

Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

Introduction to Python & Jupyter Notebook

Basic Python 6 – Loops

- For loops with list
- Noticed that we use the same keyword which is **in**. This keyword is quite convenient to use.

```
In [33]: items = ["orange", "apple", "mango", "papaya","carrot", "butterhead"]

         for barang in items:
             print(barang)
```

```
orange
apple
mango
papaya
carrot
butterhead
```

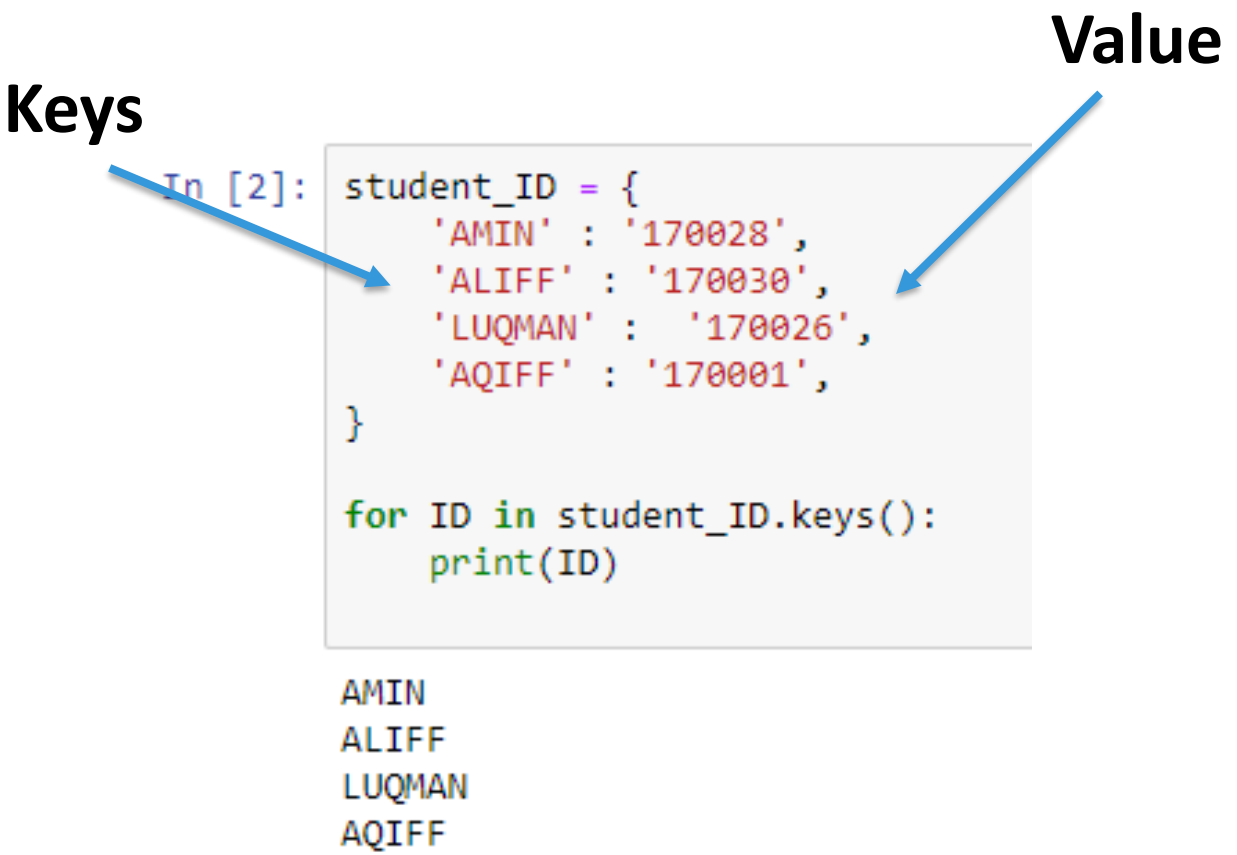
```
In [34]: print(items[0])
         print(items[1])
         print(items[2])
         print(items[3])
         print(items[4])
         print(items[5])
```

```
orange
apple
mango
papaya
carrot
butterhead
```

Introduction to Python & Jupyter Notebook

Basic Python 6 – Loops

- For loops with dictionary
- Previously we learnt about the structure of a python dictionary. By now, you should understand the concept of keys and values.
  Let's print the keys of a dictionary.



Introduction to Python & Jupyter Notebook

Basic Python 6 – Loops

- For loops with string
- We can also print or get/access every single character in a string.

```
In [4]: name = 'AMIRUL ASHRAFF'

        for character in name:
            print(character)
```

```
A
M
I
R
U
L

A
S
H
R
A
F
F
```

Introduction to Python & Jupyter Notebook

Basic Python 6 – While loops

- **While** loop has the same functionality as **for** loops. However, as the name suggests, **while** loop is executed only if the **condition is True**.
- Let's see the example below. **i < 5** is the condition set for this **while** loop. We first **initialized variable i = 0**. With every iteration, **i will be printed** and **its value will increase by 1** (increment by 1). The iteration stops until value of i is bigger than 5.

```
In [7]: i = 0
        while (i < 5):
            print(i)
            i = i + 1
```

```
0
1
2
3
4
```

DATA ANALYSIS WITH PANDAS

Module 2 : Pandas Series

Pandas Series

What is Pandas?

- Pandas is a software library written for the Python programming language for data manipulation and analysis.
- First, we need to **import the library** before we can use it. The simple command on how to do this:

```
In [4]: import pandas as pd
```

- **as** keyword is used to shorten the library name by giving it an alias. **pd** is the widely used alias for pandas among Python users.
- Let's create our first pandas Series object from a dataset. Even though Pandas Series can **read all kinds of data,** we have to make sure to keep **data consistency.**
- Here, we have a list of ice cream flavors and will pass it to pandas library to create Pandas Series.

Pandas Series

Pandas : Dtype

- **Dtype** means the **data type for a Series** . **Object** indicates that the Series is **String** type. Notice the numbers generated on the left side? These are the **indexes of each of the element in the Series.**
- The **difference** between the indexes in **Pandas Series** and the ones in **Python list** is that they do not have to be **numeric**. We will learn how to change the index and access it using .loc function in the next few lessons.

```
In [3]: ice_cream = ["Chocolate", "Banana", "Vanilla", "Strawberry"]

        pd.Series(ice_cream)

Out[3]: 0     Chocolate
        1        Banana
        2       Vanilla
        3    Strawberry
        dtype: object
```

Pandas Series

Pandas : Dtype

- Here, we created a new list with different data type and then subsequently created a Pandas Series from that list.
- Notice that the dtype is different from the previous one. It is shown as **int64** indicating our data type is now **Integer.**
- *Hint*: Keep in mind that the index for list and Pandas Series starts with 0. Hence, the last number will always be less than the total length of the list.

```
In [4]: lottery = [34,74,12,98,19]

        pd.Series(lottery)

Out[4]: 0    34
        1    74
        2    12
        3    98
        4    19
        dtype: int64
```

Pandas Series

Pandas : Dtype

- If we create a Pandas Series from a python dictionary, we can see the **difference in its index.** For this case, the **keys in dictionary** are taken to be the **indexes in the Pandas Series.**

```
In [6]: student_grade = {
            "Amin" : "80",
            "Senoi" : "90",
            "Danial" : "89",
            "Aqiff" : "100"
        }

        pd.Series(student_grade)

Out[6]: Amin       80
        Senoi      90
        Danial     89
        Aqiff     100
        dtype: object
```

Pandas Series

Pandas : Combine different data type into one list

- Combining different data types into a series will automatically convert it to **become object or String type.** However, bear in mind that this will cause problem if you plan to do mathematical operations on it later.
- *Hint* : "+" operator also works if we want to combine two or more lists.

```
In [5]: combine = ice_cream + lottery
        pd.Series(combine)

Out[5]: 0     Chocolate
        1        Banana
        2       Vanilla
        3    Strawberry
        4            34
        5            74
        6            12
        7            98
        8            19
        dtype: object
```

Pandas Series

Attributes and Methods in Pandas Series

- **Attribute** does not modify or manipulate the object in any way. Its purpose is to display and present information.
- **Method** perform some kind of operation, manipulation or calculation on the objects.

```
In [43]: lottery = [34,74,12,98,19]

         s = pd.Series(lottery)
         s

Out[43]: 0    34
         1    74
         2    12
         3    98
         4    19
         dtype: int64
```

Pandas Series

Attributes and Methods in Pandas Series

- **.values** attribute returns an array of values.

```
In [32]: s.values
Out[32]: array([34, 74, 12, 98, 19], dtype=int64)
```

- **.index** attribute shows us information on the index of the Series
  - **start** : the starting index of the Series
  - **stop** : the last index of the Series

```
In [33]: s.index
Out[33]: RangeIndex(start=0, stop=5, step=1)
```

- **.dtypes** attribute returns the data type of the Series

```
In [34]: s.dtypes
Out[34]: dtype('int64')
```

Pandas Series

Attributes and Methods in Pandas Series

- The main difference between attributes and methods is, **methods have parentheses / ()** at the end while attribute does not.
- **.sum()** method returns summation of all the values in the Series. Hence, we do not need to use any loop to do the calculation.
- **.product()** method will multiply all the values together in the Series.

```
In [35]: s.sum()

Out[35]: 237
```

```
In [37]: s.product()

Out[37]: 56217504
```

Pandas Series

Attributes and Methods in Pandas Series

- We can also do Mathematical operations using different methods.

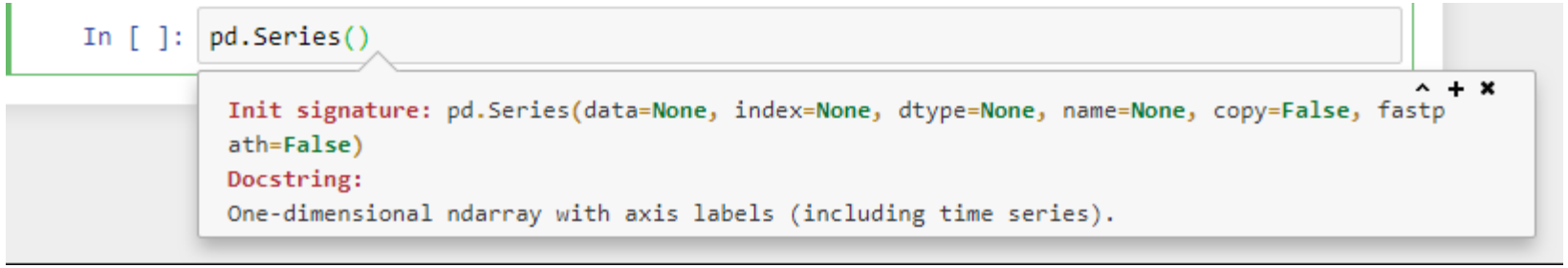
```
In [46]: s.sum() / s.count()
Out[46]: 47.4
```

- **.mean()** returns the average value of the Series

```
In [36]: s.mean()
Out[36]: 47.4
```

Pandas Series
Parameters and Arguments

- Parameter and argument are almost the same thing.
- When we want to create a method, we need to specify what parameter(s) we need. Then, when we want to call the method, we need to give argument(s) according to the parameter.
- Some parameters are set to **None**, hence there is a default value to the parameter.
- (The method below is not run. I simply write the method and **click the Shift key + Tab key** to show the details of the method.)



Pandas Series

Parameters and Arguments

- We created a Pandas Series on students' grades and set the index to their names.
- As you can see, **each name** (acting as index) **has a number** mapped to it.
- Another way of inserting the arguments is by **specifying which is the data, and which is the index**. This way, we do not have to insert the arguments in a particular order(data first, then index)

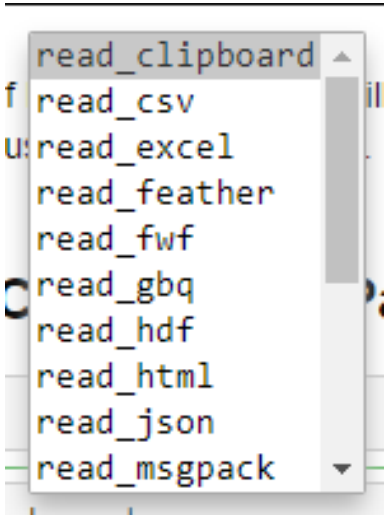
```
In [3]: student_name = ["Amin", "Senoi" ,"Danial", "Aqiff",]
        grade = [88,79,99,87]

        pd.Series(grade, student_name)

Out[3]: Amin      88
        Senoi     79
        Danial    99
        Aqiff     87
        dtype: int64
```

```
In [4]: pd.Series(data = grade, index = student_name)

Out[4]: Amin      88
        Senoi     79
        Danial    99
        Aqiff     87
        dtype: int64
```

Pandas Series

CSV file into Pandas Series

- Pandas can read many types of files; for example, JSON, Excel, CSV, and HTML.
- In the following slides, I will show you how to read CSV file and convert it into Pandas Series.



Pandas Series

CSV file into Pandas Series

- **"data/pokemon.csv"** is the location of the file.
- **usecols** indicates that we only want to import Pokemon column

```
In [22]: pd.read_csv("data/pokemon.csv", usecols=["Pokemon"]).head()
```

Out[22]:

	Pokemon
0	Bulbasaur
1	Ivysaur
2	Venusaur
3	Charmander
4	Charmeleon

- **squeeze** is set to **True** to **change Pandas DataFrame into Pandas Series**

```
In [24]: pd.read_csv("data/pokemon.csv", usecols=["Pokemon"], squeeze= True).head()
Out[24]: 0     Bulbasaur
         1       Ivysaur
         2      Venusaur
         3    Charmander
         4    Charmeleon
         Name: Pokemon, dtype: object
```

Pandas Series

CSV file into Pandas Series

```
In [3]: pokemon = pd.read_csv("data/pokemon.csv", usecols=["Pokemon"], squeeze= True)
        pokemon
```

```
Out[3]: 0       Bulbasaur
        1         Ivysaur
        2        Venusaur
        3      Charmander
        4      Charmeleon
        5       Charizard
        6        Squirtle
        7       Wartortle
        8       Blastoise
        9        Caterpie
        10        Metapod
        11     Butterfree
        12         Weedle
        13         Kakuna
        14       Beedrill
        15         Pidgey
        16      Pidgeotto
        17        Pidgeot
        18        Rattata
        19       Raticate
        20        Spearow
        21         Fearow
        22          Ekans
        23          Arbok
        24        Pikachu
        25         Raichu
```

```
In [5]: google = pd.read_csv("data/google_stock_price.csv", squeeze = True)
        google
```

```
Out[5]: 0       50.12
        1       54.10
        2       54.65
        3       52.38
        4       52.95
        5       53.90
        6       53.02
        7       50.95
        8       51.13
        9       50.07
        10      50.70
        11      49.95
        12      50.74
        13      51.10
        14      51.10
        15      52.61
        16      53.70
        17      55.69
        18      55.94
```

Pandas Series
Python Built-in Function

- The Python interpreter has a number of functions built into it that are always available.
  - **len()** : returns the total elements in a list
  - **type()** : returns the type of list of elements
  - **min()** : returns the minimum value from the list
  - **max()** : returns the maximum value from the list

```
In [11]: len(pokemon) , len(google)
Out[11]: (721, 3012)

In [12]: type(pokemon)
Out[12]: pandas.core.series.Series

In [25]: max(google)
Out[25]: 782.22

In [26]: min(pokemon)
Out[26]: 'Abomasnow'
```

Pandas Series

Pandas Series Attributes

- **.is_unique** attribute returns a Boolean value. True if there are no duplicates, False if there are duplicate values in the Series.
- For **Pokemon Series**, is_unique attribute returns **True** meaning every single **value in the Series is unique**. There is no pokemon with the same name.
- Google Series **has duplicates** because there are some days that have the same stock value.

```
In [38]: pokemon.is_unique
Out[38]: True
```

```
In [28]: google.is_unique
Out[28]: False
```

Pandas Series

Pandas Series Attributes

- **.ndim** attribute returns the dimension of the Series. In some cases, we need to create multidimensional Series.
- **.shape** attribute returns the size of the Series in tuple data type.
  - Google has 3012 rows and 1 column

```
In [30]: google.ndim
Out[30]: 1
```

```
In [31]: google.shape
Out[31]: (3012,)
```

- **.size** attribute gives information about the total number of cells in the Series. (Keep in mind that it will also count the null values).

```
In [33]: google.size
Out[33]: 3012
```

Pandas Series
Pandas Series Methods

- **.sort_values()** returns sorted Pandas Series objects.
- *Hint* : **Methods Chaining** is a style of invoking multiple method calls sequentially. For instance, after calling .sort_values method, we can call .head() method.

```
In [40]: pokemon.sort_values().head()

Out[40]: 459    Abomasnow
         62          Abra
         358        Absol
         616     Accelgor
         680    Aegislash
         Name: Pocket Monsters, dtype: object
```

```
In [41]: pokemon.sort_values(ascending=False).head()

Out[41]: 717     Zygarde
         633    Zweilous
         40        Zubat
         569       Zorua
         570     Zoroark
         Name: Pocket Monsters, dtype: object
```

Pandas Series

Pandas Series Methods

- If we want to get the highest stock price in Google Series, we can use either of the methods below.

```
In [44]: google.max()

Out[44]: 782.22

In [45]: google.sort_values(ascending=False).head(1)

Out[45]: 3011    782.22
         Name: Stock Price, dtype: float64
```

- **inplace parameter** : overwrites the original variable with the new result.

```
In [47]: google.head(3)

Out[47]: 0    50.12
         1    54.10
         2    54.65
         Name: Stock Price, dtype: float64

In [48]: google.sort_values(ascending=False, inplace=True)

In [49]: google.head(3)

Out[49]: 3011    782.22
         2859    776.60
         3009    773.18
         Name: Stock Price, dtype: float64
```

Pandas Series
.head() and .tail() methods

- Both of these methods returns a copy of objects at certain rows in the Pandas Series.
- **.head() method** will take the first few objects while **.tail() method** will take the last few objects in the Series.
- By default, the methods will take the first 5 or last 5 unless an argument is specified.

```
In [31]: pokemon.head()

Out[31]: 0     Bulbasaur
         1       Ivysaur
         2      Venusaur
         3    Charmander
         4    Charmeleon
         Name: Pokemon, dtype: object
```

```
In [32]: pokemon.tail()

Out[32]: 716      Yveltal
         717      Zygarde
         718      Diancie
         719        Hoopa
         720    Volcanion
         Name: Pokemon, dtype: object
```

```
In [33]: pokemon.head(10)

Out[33]: 0     Bulbasaur
         1       Ivysaur
         2      Venusaur
         3    Charmander
         4    Charmeleon
         5     Charizard
         6      Squirtle
         7     Wartortle
         8     Blastoise
         9      Caterpie
         Name: Pokemon, dtype: object
```

Pandas Series

.head() and .tail() methods

- **.sort_index() method** : sort the list based on the index.
- If we sort the Pokemon Series according to its value, we can see that the order of the index number has changed.

```
In [9]: pokemon.sort_values(ascending=False, inplace=True)
        pokemon.head()

Out[9]: 717     Zygarde
        633    Zweilous
        40        Zubat
        569       Zorua
        570     Zoroark
        Name: Pokemon, dtype: object
```

- To sort the series again based on the index number, we can use **.sort_index()** method.

```
In [10]: pokemon.sort_index(inplace=True)
         pokemon.head()

Out[10]: 0     Bulbasaur
         1       Ivysaur
         2      Venusaur
         3    Charmander
         4    Charmeleon
         Name: Pokemon, dtype: object
```

Pandas Series

Pandas *in* keyword

- return a Boolean value after checking the values in the list. It will return **True** if the **element exists** in the list, and False if it does not.

```
In [12]: 3 in [1,2,3,4,5]
Out[12]: True

In [13]: pokemon.head()
Out[13]: 0     Bulbasaur
         1       Ivysaur
         2      Venusaur
         3    Charmander
         4    Charmeleon
         Name: Pokemon, dtype: object
```

- For Pandas Series, by default, *in* keyword will **check** the Series' index. If we want to check the values, then we need to specify it.

```
In [15]: "Bulbasaur" in pokemon
Out[15]: False

In [14]: "Bulbasaur" in pokemon.values
Out[14]: True
```

Pandas Series

Extract Values by Index Number Position

- Series works like a list. We can access specific data using bracket, [ ] notation. Let's access the first and last data of Pokemon.

```
In [16]: pokemon.head()

Out[16]: 0     Bulbasaur
         1       Ivysaur
         2      Venusaur
         3    Charmander
         4    Charmeleon
         Name: Pokemon, dtype: object
```

```
In [17]: pokemon[0]

Out[17]: 'Bulbasaur'
```

```
In [18]: pokemon.tail()

Out[18]: 716      Yveltal
         717      Zygarde
         718      Diancie
         719        Hoopa
         720    Volcanion
         Name: Pokemon, dtype: object
```

```
In [19]: pokemon[720]

Out[19]: 'Volcanion'
```

Pandas Series

Extract Values by Index Number Position

- Access data in a certain range by using colon (:). For example, let's display the pokemon name between number 10 to 20.
- *Hint*: Always add 1 to the end number. Like in this example, we would like to end at number 20. So we have to type 21.

```
In [27]: pokemon[10:21]

Out[27]: 10       Metapod
         11    Butterfree
         12        Weedle
         13        Kakuna
         14      Beedrill
         15        Pidgey
         16     Pidgeotto
         17       Pidgeot
         18       Rattata
         19      Raticate
         20       Spearow
         Name: Pokemon, dtype: object
```

- We can also access the Series using a negative number. Negative number means the counting starts backward.
- Here, we are displaying the last 10 values in the Series.

```
In [33]: pokemon[-10:]

Out[33]: 711     Bergmite
         712      Avalugg
         713       Noibat
         714      Noivern
         715      Xerneas
         716      Yveltal
         717      Zygarde
         718      Diancie
         719        Hoopa
         720    Volcanion
         Name: Pokemon, dtype: object
```

Pandas Series

Extract Series Values by Index Label

- To do this, we need to change the index from number to the Pokemon name using **index_col** parameter.

```
In [30]: pokemon = pd.read_csv("data/pokemon.csv", index_col = "Pokemon", squeeze=True)
         pokemon.head(3)

Out[30]: Pokemon
         Bulbasaur    Grass
         Ivysaur      Grass
         Venusaur     Grass
         Name: Type, dtype: object

In [20]: pokemon[["Bulbasaur" ,"Ditto", "Meowth"]]

Out[20]: Pokemon
         Bulbasaur     Grass
         Ditto        Normal
         Meowth       Normal
         Name: Type, dtype: object
```

Pandas Series

Math Methods on Series Object

- There are many mathematical methods that we can use to help ease our works. e.g., median()

```
In [50]: google.median()
Out[50]: 283.315
```

- **.describe()** method gives a brief information of the Series.
  - **count**: total number of elements
  - **mean** : the average number of the Series
  - **std** : Standard Deviation
  - **min** : the smallest value in the Series
  - **max** : the highest value in the Series
  - **25%** : 1st quartile.
  - **50%** : 2nd quartile/ median.
  - **75%** : 3rd quartile

```
In [57]: google.describe()
Out[57]: count    3012.000000
         mean      334.310093
         std       173.187205
         min        49.950000
         25%       218.045000
         50%       283.315000
         75%       443.000000
         max       782.220000
         Name: Stock Price, dtype: float64
```

Pandas Series

.Pandas Series Methods

- .idxmax() and .idxmin() Methods
  - return the position index of the max/min value

```
In [85]: google.min()
Out[85]: 49.95

In [87]: minIndex = google.idxmin()
         google[minIndex]
Out[87]: 49.95
```

- .value_counts() Method
  - Returns a new Series of unique counts on the Series. For example, I want to know how many Fire and Water Pokemon.

```
In [92]: pokemon = pd.read_csv("data/pokemon.csv", squeeze = True, index_col= "Pokemon" )
         pokemon.head()
Out[92]: Pokemon
         Bulbasaur     Grass
         Ivysaur       Grass
         Venusaur      Grass
         Charmander     Fire
         Charmeleon     Fire
         Name: Type, dtype: object
```

```
In [93]: pokemon.value_counts()
Out[93]: Water       105
         Normal       93
         Grass        66
         Bug          63
         Psychic      47
         Fire         47
         Rock         41
         Electric     36
         Ground       30
         Poison       28
         Dark         28
```

Pandas Series

.Pandas Series Methods

- **.apply() method** apply changes on every value in the Series based on the passed method.
- For instance, I want to set a threshold on the google stock performance so I created a method as shown.
- Then, I invoke that method on the Series' elements using the **apply() method.**

```
In [112]: def performace_indicator(number):
              if number < 300:
                  return "OK"
              elif number >= 300 and number <= 650:
                  return "Quite good"
              else: return "Increadible!"

In [103]: google.apply(performace_indicator).head()

Out[103]: 0    OK
          1    OK
          2    OK
          3    OK
          4    OK
          Name: Stock Price, dtype: object

In [104]: google.apply(performace_indicator).tail()

Out[104]: 3007    Increadible!
          3008    Increadible!
          3009    Increadible!
          3010    Increadible!
          3011    Increadible!
          Name: Stock Price, dtype: object
```

DATA ANALYSIS WITH PANDAS

Module 3 : Pandas DataFrame

Pandas DataFrame

Introduction

- Pandas **Series** is a one-dimensional array. Meanwhile **Pandas DataFrame** is a **two-dimensional, size-mutable**, potentially heterogeneous tabular data structure with labelled axes.
- Two-dimensional means it has rows and columns. To extract specific information we need to have both information on row and column.
- Below is the example of DataFrame

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN
3	R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1148640.0
4	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN	5000000.0
5	Amir Johnson	Boston Celtics	90.0	PF	29.0	6-9	240.0	NaN	12000000.0
6	Jordan Mickey	Boston Celtics	55.0	PF	21.0	6-8	235.0	LSU	1170960.0

Pandas DataFrame

Introduction

- Now we are going to read the **nba.csv file.**
- The file contains information about the player in the 2015/2016 season.
- NaN value means blank or null value. Automatically assigned when it finds a missing value.

```
In [3]: nba = pd.read_csv("data/nba.csv")
        nba
```

Out[3]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN
3	R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1148640.0
4	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN	5000000.0
5	Amir Johnson	Boston Celtics	90.0	PF	29.0	6-9	240.0	NaN	12000000.0
6	Jordan Mickey	Boston Celtics	55.0	PF	21.0	6-8	235.0	LSU	1170960.0
7	Kelly Olynyk	Boston Celtics	41.0	C	25.0	7-0	238.0	Gonzaga	2165160.0
8	Terry Rozier	Boston Celtics	12.0	PG	22.0	6-2	190.0	Louisville	1824360.0
9	Marcus Smart	Boston Celtics	36.0	PG	22.0	6-4	220.0	Oklahoma State	3431040.0
10	Jared Sullinger	Boston Celtics	7.0	C	24.0	6-9	260.0	Ohio State	2569260.0
11	Isaiah Thomas	Boston Celtics	4.0	PG	27.0	5-9	185.0	Washington	6912869.0

Pandas DataFrame

Introduction

- The structure of a DataFrame is almost like a Series. It will only show the first 30 rows and last 30 rows. In between that, it will truncate the table and replace with **...** symbol.
- Pandas DataFrame automatically generates indexes on the left side.
- At the bottom of the table, we can see the total number of rows and columns. In this table, there are 458 rows and 9 columns.

454	Raul Neto	Utah Jazz	25.0	PG	24.0	6-1	179.0	NaN	900000.0
455	Tibor Pleiss	Utah Jazz	21.0	C	26.0	7-3	256.0	NaN	2900000.0
456	Jeff Withey	Utah Jazz	24.0	C	26.0	7-0	231.0	Kansas	947276.0
457	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

458 rows × 9 columns

Pandas DataFrame

Methods and Attributes in Pandas DataFrame

- de

- **.index** attribute returns information about the object's index.

```
In [25]: nba.index
Out[25]: RangeIndex(start=0, stop=458, step=1)
```

- **.values** attribute returns the values of the object.

```
In [9]: nba.values
Out[9]: array([['Avery Bradley', 'Boston Celtics', 0.0, ..., 180.0, 'Texas',
                7730337.0],
               ['Jae Crowder', 'Boston Celtics', 99.0, ..., 235.0, 'Marquette',
                6796117.0],
               ['John Holland', 'Boston Celtics', 30.0, ..., 205.0,
                'Boston University', nan],
               ...,
               ['Tibor Pleiss', 'Utah Jazz', 21.0, ..., 256.0, nan, 2900000.0],
               ['Jeff Withey', 'Utah Jazz', 24.0, ..., 231.0, 'Kansas', 947276.0],
               [nan, nan, nan, ..., nan, nan, nan]], dtype=object)
```

Pandas DataFrame

Methods and Attributes in Pandas DataFrame

- **.shape** attribute returns the number of rows and columns of the object.

```
In [10]: nba.shape
Out[10]: (458, 9)
```

- **.dtypes** attributes returns the data type for each columns.
  - *Hint:* object means string

```
In [26]: nba.dtypes
Out[26]: Name          object
         Team          object
         Number       float64
         Position      object
         Age          float64
         Height        object
         Weight       float64
         College       object
         Salary       float64
         dtype: object
```

Pandas DataFrame

Methods and Attributes in Pandas DataFrame

- **.columns** returns a list of column names

```
In [27]: nba.columns

Out[27]: Index(['Name', 'Team', 'Number', 'Position', 'Age', 'Height', 'Weight',
                'College', 'Salary'],
               dtype='object')
```

- **.axes** attribute returns information on our axes, which are index and column names.

```
In [31]: nba.axes

Out[31]: [RangeIndex(start=0, stop=458, step=1),
          Index(['Name', 'Team', 'Number', 'Position', 'Age', 'Height', 'Weight',
                 'College', 'Salary'],
                dtype='object')]
```

- **.head()** : returns first N row(s) of the DataFrame. By default, N=5.

```
In [4]: nba.head()
```

Out[4]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN
3	R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1148640.0
4	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN	5000000.0

Pandas DataFrame

Methods and Attributes in Pandas DataFrame

- **.tail()** : returns last N row(s) of the DataFrame. Default value of N is 5.

```
In [6]: nba.tail(3)
```

Out[6]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
455	Tibor Pleiss	Utah Jazz	21.0	C	26.0	7-3	256.0	NaN	2900000.0
456	Jeff Withey	Utah Jazz	24.0	C	26.0	7-0	231.0	Kansas	947276.0
457	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- **.get_dtypes_counts** returns information on our columns data type. In this example, there are 4 columns with data type float and 5 with string.

```
In [34]: nba.get_dtype_counts()
```

```
Out[34]: float64    4
         object     5
         dtype: int64
```

Pandas DataFrame

Methods and Attributes in Pandas DataFrame

- **info()** method returns a summary of the DataFrame. This makes understanding the data a lot easier.
- According to the DataFrame's summary, we can see that College and Salary columns have 373 and 446 rows respectively. However, the total number of rows is actually 458. This means that these columns have some missing elements.

```
In [32]: nba.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 458 entries, 0 to 457
Data columns (total 9 columns):
Name        457 non-null object
Team        457 non-null object
Number      457 non-null float64
Position    457 non-null object
Age         457 non-null float64
Height      457 non-null object
Weight      457 non-null float64
College     373 non-null object
Salary      446 non-null float64
dtypes: float64(4), object(5)
memory usage: 32.3+ KB
```

Pandas DataFrame

Methods and Attributes in Pandas DataFrame

- **.sum()** method returns the summation of numerical columns' elements. We can choose either to sum horizontally or vertically.
- In this example, we are adding all the rows vertically. We will get the total revenue of New York which is 5475.

```
In [38]: rev = pd.read_csv("data/revenue.csv", index_col="Date")
         rev.head(3)
```

Out[38]:

Date	New York	Los Angeles	Miami
1/1/16	985	122	499
1/2/16	738	788	534
1/3/16	14	20	933

```
In [39]: rev.sum()
```

```
Out[39]: New York       5475
         Los Angeles    5134
         Miami          5641
         dtype: int64
```

Pandas DataFrame

Methods and Attributes in Pandas DataFrame

- When we use **parameter axis** and give argument of 1 or columns, it does summation operation over the column axis.

```
In [185]: rev.sum(axis="columns")

Out[185]: Date
          1/1/16     1606
          1/2/16     2060
          1/3/16      967
          1/4/16     2519
          1/5/16      438
          1/6/16     1935
          1/7/16     1234
          1/8/16     2313
          1/9/16     2623
          1/10/16     555
          dtype: int64
```

```
In [45]: rev.sum(axis=1)

Out[45]: Date
         1/1/16     3212
         1/2/16     4120
         1/3/16     1934
         1/4/16     5038
         1/5/16      876
         1/6/16     3870
         1/7/16     2468
         1/8/16     4626
         1/9/16     5246
         1/10/16    1110
         dtype: int64
```

```
In [184]: rev = pd.read_csv("data/revenue.csv", index_col="Date")
          rev.head()
```

Out[184]:

Date	New York	Los Angeles	Miami
1/1/16	985	122	499
1/2/16	738	788	534
1/3/16	14	20	933
1/4/16	730	904	885
1/5/16	114	71	253

Pandas DataFrame

Methods and Attributes in Pandas DataFrame

- We can also create a new column in the DataFrame to store the results of **sum() method.**

```
In [16]: rev["Sum by day"] = rev.sum(axis=1)

In [17]: rev.head()
```

Out[17]:

Date	New York	Los Angeles	Miami	Sum by day
1/1/16	985	122	499	1606
1/2/16	738	788	534	2060
1/3/16	14	20	933	967
1/4/16	730	904	885	2519
1/5/16	114	71	253	438

Pandas DataFrame

Extract One Column from DataFrame

- To extract values of a specific column, you can call the column's name on the DataFrame variable. For example, **nba.Name** will return all the values in the "Name" column of "nba" DataFrame.
- *Hint*: Always remember the column name is **Case Sensitive**. Capital and small letters will give different results.

```
In [18]: nba = pd.read_csv("data/nba.csv")
         nba.head(3)
```

Out[18]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN

```
In [19]: nba.Name.head()
```

```
Out[19]: 0    Avery Bradley
         1      Jae Crowder
         2     John Holland
         3      R.J. Hunter
         4    Jonas Jerebko
         Name: Name, dtype: object
```

Pandas DataFrame

Extract One Column from DataFrame

- Another way is to add square brackets “[ ]” around the column’s name.
- It is often better to use the second way as the first way may **not work** well with **column names that contain spaces**. For example, "Player Name".
- When extracting a specific column, the result will be returned in Pandas Series. To make it return in Pandas DataFrame, we can add another "[ ]" bracket around the column name.

```
In [20]: nba["Name"].head()
Out[20]: 0    Avery Bradley
         1      Jae Crowder
         2     John Holland
         3      R.J. Hunter
         4    Jonas Jerebko
         Name: Name, dtype: object
```

```
In [21]: nba[["Name"]].head()
```

Out[21]:

	Name
0	Avery Bradley
1	Jae Crowder
2	John Holland
3	R.J. Hunter
4	Jonas Jerebko

Pandas DataFrame

Select Two or More Columns in Data Frame

- We can extract values of more than one column using the same way by specifying **all the columns we want** inside the square brackets. For example, if we want to extract “Name” and “Salary” column.
- *Hint* : The **order** of how they are written is **not important**

```
In [23]: nba[["Name", "Salary"]].head()
```

Out[23]:

	Name	Salary
0	Avery Bradley	7730337.0
1	Jae Crowder	6796117.0
2	John Holland	NaN
3	R.J. Hunter	1148640.0
4	Jonas Jerebko	5000000.0

```
In [24]: nba[[ "Team" , "Name", "College"]].head()
```

Out[24]:

	Team	Name	College
0	Boston Celtics	Avery Bradley	Texas
1	Boston Celtics	Jae Crowder	Marquette
2	Boston Celtics	John Holland	Boston University
3	Boston Celtics	R.J. Hunter	Georgia State
4	Boston Celtics	Jonas Jerebko	NaN

Pandas DataFrame

Add a new column to DataFrame

- We can also use this way to add column which does not exist in the DataFrame and assign values into it.
- For example, here we are adding a new column named “Sport” into the DataFrame and setting “Basketball” as its values.

```
In [26]: nba["Sport"] = "Basketball"

In [27]: nba.head()
```

Out[27]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary	Sport
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0	Basketball
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0	Basketball
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN	Basketball
3	R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1148640.0	Basketball
4	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN	5000000.0	Basketball

Pandas DataFrame

Add a new column to DataFrame : **insert() method**

- **.insert()** method also does the same job. However, we can be more specific on the location of our new column.

- **Parameters:**
  - **loc** : location of the new column
  - **column** : name for the new column
  - **value** : the value for the new column

```
In [29]: nba.insert(loc = 3, column = "Sport" , value = "Basketball")

In [30]: nba.head(3)
```

Out[30]:

	Name	Team	Number	Sport	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	Basketball	PG	25.0	6-2	180.0	Texas	7730337.0
1	Jae Crowder	Boston Celtics	99.0	Basketball	SF	25.0	6-6	235.0	Marquette	6796117.0
2	John Holland	Boston Celtics	30.0	Basketball	SG	27.0	6-5	205.0	Boston University	NaN

Pandas DataFrame

Broadcasting Operations

- Pandas have methods to compute operation on every row of every column.
- Their advantage is they come with support to handle the NaN value and will not prompt error. We also do **not have to use for loop to iterate every row.**
- **.add()** : add each row with a specified value

```
In [31]: nba["Age"].add(5).head()

Out[31]: 0    30.0
         1    30.0
         2    32.0
         3    27.0
         4    34.0
         Name: Age, dtype: float64
```

- **.sub()** : subtract each row with a specified value

```
In [32]: nba["Salary"].sub(100000).head(10)

Out[32]: 0     7630337.0
         1     6696117.0
         2           NaN
         3     1048640.0
         4     4900000.0
         5    11900000.0
         6     1070960.0
         7     2065160.0
         8     1724360.0
         9     3331040.0
         Name: Salary, dtype: float64
```

Pandas DataFrame

Broadcasting Operations

- **.mul()** : do multiplication operation for each row
  - Change the weight to Kilogram, then assign the result to a new column
  - *Hint* : 1 lbs = 0.453592 kg

```
In [33]: nba["Weight in Kilogram"] = nba["Weight"].mul(0.453592)

In [34]: nba.head()
```

Out[34]:

	Name	Team	Number	Sport	Position	Age	Height	Weight	College	Salary	Weight in Kilogram
0	Avery Bradley	Boston Celtics	0.0	Basketball	PG	25.0	6-2	180.0	Texas	7730337.0	81.646560
1	Jae Crowder	Boston Celtics	99.0	Basketball	SF	25.0	6-6	235.0	Marquette	6796117.0	106.594120
2	John Holland	Boston Celtics	30.0	Basketball	SG	27.0	6-5	205.0	Boston University	NaN	92.986360
3	R.J. Hunter	Boston Celtics	28.0	Basketball	SG	22.0	6-5	185.0	Georgia State	1148640.0	83.914520
4	Jonas Jerebko	Boston Celtics	8.0	Basketball	PF	29.0	6-10	231.0	NaN	5000000.0	104.779752

Pandas DataFrame

value_counts() method

- This method returns an object containing counts of **unique values.** The resulting object will be in descending order so that the **first element is the most frequently-occurring element.** By default, it will exclude NA values.

- Information we can gain from value counts is
  - **SG** or Shooting Guard is the most popular position among the players.
  - **C** or Center is the least popular
  - **New Orleans Pelicans** hold the most players

```
In [36]: nba["Position"].value_counts()

Out[36]: SG    102
         PF    100
         PG     92
         SF     85
         C      78
         Name: Position, dtype: int64
```

Pandas DataFrame

Drop Rows with Null Values

- **dropna()** removes row, which contains null or NaN value. The return value is DataFrame object
  - **how**: the default value is "any" means if there is at least one Null value, the row will be removed. If how = "all “, only row with all its values are null will be removed
  - **inplace** : store permanent changes
  - **axis** : default value is 0. If we change to 1, it will remove columns that contain any null value.
  - **subset** : choose which column(s) contain null values. By default it will check all the columns.

Pandas DataFrame

Drop Rows with Null Values

```
In [38]: nba = pd.read_csv("data/nba.csv")
         nba.tail()
```

Out[38]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
453	Shelvin Mack	Utah Jazz	8.0	PG	26.0	6-3	203.0	Butler	2433333.0
454	Raul Neto	Utah Jazz	25.0	PG	24.0	6-1	179.0	NaN	900000.0
455	Tibor Pleiss	Utah Jazz	21.0	C	26.0	7-3	256.0	NaN	2900000.0
456	Jeff Withey	Utah Jazz	24.0	C	26.0	7-0	231.0	Kansas	947276.0
457	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

```
In [39]: nba.dropna(how="all", inplace=True)
```

```
In [40]: nba.tail()
```

Out[40]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
452	Trey Lyles	Utah Jazz	41.0	PF	20.0	6-10	234.0	Kentucky	2239800.0
453	Shelvin Mack	Utah Jazz	8.0	PG	26.0	6-3	203.0	Butler	2433333.0
454	Raul Neto	Utah Jazz	25.0	PG	24.0	6-1	179.0	NaN	900000.0
455	Tibor Pleiss	Utah Jazz	21.0	C	26.0	7-3	256.0	NaN	2900000.0
456	Jeff Withey	Utah Jazz	24.0	C	26.0	7-0	231.0	Kansas	947276.0

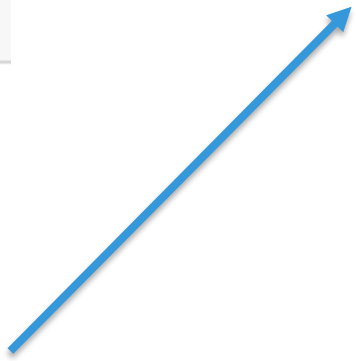
Pandas DataFrame

Drop Rows with Null Values

- Remove all rows that contain at least one Null value. Then, check the summary of the new data using the .info() method.

```
nba.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 458 entries, 0 to 457
Data columns (total 9 columns):
Name        457 non-null object
Team        457 non-null object
Number      457 non-null float64
Position    457 non-null object
Age         457 non-null float64
Height      457 non-null object
Weight      457 non-null float64
College     373 non-null object
Salary      446 non-null float64
dtypes: float64(4), object(5)
memory usage: 32.3+ KB
```



```
In [41]: nba.dropna(inplace=True)
```

```
In [42]: nba.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 364 entries, 0 to 456
Data columns (total 9 columns):
Name        364 non-null object
Team        364 non-null object
Number      364 non-null float64
Position    364 non-null object
Age         364 non-null float64
Height      364 non-null object
Weight      364 non-null float64
College     364 non-null object
Salary      364 non-null float64
dtypes: float64(4), object(5)
memory usage: 28.4+ KB
```

Pandas DataFrame

Remove column that has null value

- First, let's remove rows which contain all null values. Next, use dropna() method again with parameter **axis = 1.**

- As a result, we can see that **Salary and College column is dropped** because these two columns have null values.

In [43]:
```
nba = pd.read_csv("data/nba.csv")
nba.dropna(how="all", inplace=True)
nba.head(3)
```

Out[43]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN

In [44]:
```
nba.dropna(axis=1).head()
```

Out[44]:

	Name	Team	Number	Position	Age	Height	Weight
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0
3	R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0
4	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0

Pandas DataFrame

Fill in the Null Values with .fillna() method

- Fill NA/NaN values using the specified method.
- We need **to select a specific column** and **then apply fillna()** method on it. If we apply fillna() method on the whole DataFrame, the result will not be good as not all columns have the same data type.
- For instance, I want **to replace the NaN value in Salary column with the mean value.** 0 might seems illogical as no player is willing to play for free.

In [197]:
```
nba = pd.read_csv("data/nba.csv")
nba.dropna(how="all", inplace=True)
mask = nba["Salary"].isna()
nba[mask].head()
```

Out[197]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN
46	Elton Brand	Philadelphia 76ers	42.0	PF	37.0	6-9	254.0	Duke	NaN
171	Dahntay Jones	Cleveland Cavaliers	30.0	SG	35.0	6-6	225.0	Duke	NaN
264	Jordan Farmar	Memphis Grizzlies	4.0	PG	29.0	6-2	180.0	UCLA	NaN
269	Ray McCallum	Memphis Grizzlies	5.0	PG	24.0	6-3	190.0	Detroit	NaN

In [193]:
```
mean_Salary = nba["Salary"].mean()
mean_Salary
```

Out[193]: 4842684.105381166

In [194]:
```
nba["Salary"].fillna(mean_Salary, inplace=True)
```

In [195]:
```
nba["College"].fillna("No College", inplace=True)
```

Pandas DataFrame

Fill in the Null Values with .fillna() method

```
nba[mask].head()
```

Out[197]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN
46	Elton Brand	Philadelphia 76ers	42.0	PF	37.0	6-9	254.0	Duke	NaN
171	Dahntay Jones	Cleveland Cavaliers	30.0	SG	35.0	6-6	225.0	Duke	NaN
264	Jordan Farmar	Memphis Grizzlies	4.0	PG	29.0	6-2	180.0	UCLA	NaN
269	Ray McCallum	Memphis Grizzlies	5.0	PG	24.0	6-3	190.0	Detroit	NaN

In [202]:
```
nba[mask].head()
```

Out[202]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	4.842684e+06
46	Elton Brand	Philadelphia 76ers	42.0	PF	37.0	6-9	254.0	Duke	4.842684e+06
171	Dahntay Jones	Cleveland Cavaliers	30.0	SG	35.0	6-6	225.0	Duke	4.842684e+06
264	Jordan Farmar	Memphis Grizzlies	4.0	PG	29.0	6-2	180.0	UCLA	4.842684e+06
269	Ray McCallum	Memphis Grizzlies	5.0	PG	24.0	6-3	190.0	Detroit	4.842684e+06

Pandas DataFrame

The .astype() method

- **Change a column data type.** This method requires the column to be filled and has **no NaN value**.
- Jersey Number in Number column are supposed to be integer type. It is weird to see a jersey with a point. Let's change the data type to integer.
- **astype()** method does not have inplace parameter. Hence, we need to **reassign the nba variable.**

```
In [55]: nba.dtypes
         nba.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 364 entries, 0 to 363
Data columns (total 10 columns):
index       364 non-null int64
Name        364 non-null object
Team        364 non-null object
Number      364 non-null float64
Position    364 non-null object
Age         364 non-null float64
Height      364 non-null object
Weight      364 non-null float64
College     364 non-null object
Salary      364 non-null float64
dtypes: float64(4), int64(1), object(5)
memory usage: 28.5+ KB
```

BEFORE

```
In [58]: nba["Team"] = nba["Team"].astype("category")

In [59]: nba.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 364 entries, 0 to 363
Data columns (total 10 columns):
index       364 non-null int64
Name        364 non-null object
Team        364 non-null category
Number      364 non-null int32
Position    364 non-null object
Age         364 non-null float64
Height      364 non-null object
Weight      364 non-null float64
College     364 non-null object
Salary      364 non-null float64
dtypes: category(1), float64(3), int32(1), int64(1), object(4)
memory usage: 26.1+ KB
```

AFTER

Pandas DataFrame

Sort DataFrame with the .sort_values() Method

- Sort all columns based on a specific column.
  - **by** : using column name as references
  - **ascending** : default value is True. If set to False, it will sort in descending order.
  - **na_position** : default value is **"last"** meaning the NaN value will be placed in the last row. We also can change it to **“first”.**

```
In [60]: nba = pd.read_csv("data/nba.csv")
         nba.head(3)
```

Out[60]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN

```
In [61]: nba.sort_values(by = "Name").head()
```

Out[61]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
152	Aaron Brooks	Chicago Bulls	0.0	PG	31.0	6-0	161.0	Oregon	2250000.0
356	Aaron Gordon	Orlando Magic	0.0	PF	20.0	6-9	220.0	Arizona	4171680.0
328	Aaron Harrison	Charlotte Hornets	9.0	SG	21.0	6-6	210.0	Kentucky	525093.0
404	Adreian Payne	Minnesota Timberwolves	33.0	PF	25.0	6-10	237.0	Michigan State	1938840.0
312	Al Horford	Atlanta Hawks	15.0	C	30.0	6-10	245.0	Florida	12000000.0

Pandas DataFrame

Sort DataFrame with the .sort_values() Method

- Now we know, Player Kobe Bryant has the highest salary.

```
In [62]: nba.sort_values(by = "Salary", ascending=False).head()
```

Out[62]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
109	Kobe Bryant	Los Angeles Lakers	24.0	SF	37.0	6-6	212.0	NaN	25000000.0
169	LeBron James	Cleveland Cavaliers	23.0	SF	31.0	6-8	250.0	NaN	22970500.0
33	Carmelo Anthony	New York Knicks	7.0	SF	32.0	6-8	240.0	Syracuse	22875000.0
251	Dwight Howard	Houston Rockets	12.0	C	30.0	6-11	265.0	NaN	22359364.0
339	Chris Bosh	Miami Heat	1.0	PF	32.0	6-11	235.0	Georgia Tech	22192730.0

Pandas DataFrame

Sort_values() with two or more column references

- This can be achieved by listing all the columns' names we want inside the **by parameter**.
- We can also specify how each of the columns should be sorted (ascending or descending order).
- In this example, we will sort by **Team name in ascending order** and **Salary in descending order**. As a result, we can see who has the highest Salary in the first Team.

```
nba.sort_values(by = ["Team","Salary"], ascending=[True, False])
```

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
315	Paul Millsap	Atlanta Hawks	4.0	PF	31.0	6-8	246.0	Louisiana Tech	18671659.0
312	Al Horford	Atlanta Hawks	15.0	C	30.0	6-10	245.0	Florida	12000000.0
321	Tiago Splitter	Atlanta Hawks	11.0	C	31.0	6-11	245.0	NaN	9756250.0
323	Jeff Teague	Atlanta Hawks	0.0	PG	27.0	6-2	186.0	Wake Forest	8000000.0
314	Kyle Korver	Atlanta Hawks	26.0	SG	35.0	6-7	212.0	Creighton	5746479.0
320	Thabo Sefolosha	Atlanta Hawks	25.0	SF	32.0	6-7	220.0	NaN	4000000.0
319	Mike Scott	Atlanta Hawks	32.0	PF	27.0	6-8	237.0	Virginia	3333333.0
311	Kirk Hinrich	Atlanta Hawks	12.0	SG	35.0	6-4	190.0	Kansas	2854940.0
309	Kent Bazemore	Atlanta Hawks	24.0	SF	26.0	6-5	201.0	Old Dominion	2000000.0

Pandas DataFrame

.rank() method

- Compute data ranks (1 through n). Equal values are assigned a rank that is the average of the ranks of those values.
- For example, we can create a ranking of player based on their salary. Method chaining is used to compute them in one line of code.

```
In [66]: nba = pd.read_csv("data/nba.csv")
         nba.fillna(nba["Salary"].mean(), inplace=True)
         nba.head(3)
```

Out[66]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7.730337e+06
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6.796117e+06
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	4.842684e+06

```
In [67]: nba["Salary Rank"] = nba["Salary"].rank(ascending=False).astype("int")
```

```
In [68]: nba.head()
```

Out[68]:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary	Salary Rank
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7.730337e+06	97
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6.796117e+06	110
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	4.842684e+06	156
3	R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1.148640e+06	334
4	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	4.84268e+06	5.000000e+06	147

Pandas DataFrame

.rank() method

- Here we can see the players' ranks based on their salary

```
nba.sort_values("Salary Rank")
```

	Name	Team	Number	Position	Age	Height	Weight	College	Salary	Salary Rank
109	Kobe Bryant	Los Angeles Lakers	24.0	SF	37.0	6-6	212.0	4.84268e+06	25000000.0	1
169	LeBron James	Cleveland Cavaliers	23.0	SF	31.0	6-8	250.0	4.84268e+06	22970500.0	2
33	Carmelo Anthony	New York Knicks	7.0	SF	32.0	6-8	240.0	Syracuse	22875000.0	3
251	Dwight Howard	Houston Rockets	12.0	C	30.0	6-11	265.0	4.84268e+06	22359364.0	4
339	Chris Bosh	Miami Heat	1.0	PF	32.0	6-11	235.0	Georgia Tech	22192730.0	5
100	Chris Paul	Los Angeles Clippers	3.0	PG	31.0	6-0	175.0	Wake Forest	21468695.0	6
414	Kevin Durant	Oklahoma City Thunder	35.0	SF	27.0	6-9	240.0	Texas	20158622.0	7
164	Derrick Rose	Chicago Bulls	1.0	PG	27.0	6-3	190.0	Memphis	20093064.0	8
349	Dwyane Wade	Miami Heat	3.0	SG	34.0	6-4	220.0	Marquette	20000000.0	9
294	LaMarcus Aldridge	San Antonio Spurs	12.0	PF	30.0	6-11	240.0	Texas	19689000.0	11
23	Brook Lopez	Brooklyn Nets	11.0	C	28.0	7-0	275.0	Stanford	19689000.0	11

DATA ANALYSIS WITH PANDAS

Module 3 : DataFrame Part 2

Filtering and Cleaning

Pandas DataFrame

Understand the dataset

- **Problems in the dataset**
  - First Name, Gender, Senior Management, Team have missing values/NaN
  - Start Date and Last Login Time columns are not in datetime type
  - Senior Management column consists of data which are either True or False. But the column's data type is not Boolean.
  - Gender and Team columns' data type is not category.

```
import pandas as pd
```

```
employees = pd.read_csv("data/employees.csv")
employees.head()
```

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	8/6/1993	12:42 PM	97308	6.945	True	Marketing
1	Thomas	Male	3/31/1996	6:53 AM	61933	4.170	True	NaN
2	Maria	Female	4/23/1993	11:17 AM	130590	11.858	False	Finance
3	Jerry	Male	3/4/2005	1:00 PM	138705	9.340	True	Finance
4	Larry	Male	1/24/1998	4:47 PM	101004	1.389	True	Client Services

```
In [72]: employees.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 8 columns):
First Name           933 non-null object
Gender               855 non-null object
Start Date           1000 non-null object
Last Login Time      1000 non-null object
Salary               1000 non-null int64
Bonus %              1000 non-null float64
Senior Management    933 non-null object
Team                 957 non-null object
dtypes: float64(1), int64(1), object(6)
memory usage: 62.6+ KB
```

Pandas DataFrame

1. Change Start Date and Last Login Time columns into datetime data type

- Pandas has **to_datetime()** method to **change data type** of a column into datetime type.
- We simply have to reassign the result of this method to the column name so we do not have to create a new column and delete the old one.
- Notice the change of dtype after calling info() method.

```
In [73]: employees["Start Date"].head()

Out[73]: 0     8/6/1993
         1    3/31/1996
         2    4/23/1993
         3     3/4/2005
         4    1/24/1998
         Name: Start Date, dtype: object

In [74]: employees["Start Date"] = pd.to_datetime(employees["Start Date"])
         employees["Last Login Time"] = pd.to_datetime(employees["Last Login Time"])

In [75]: employees.info()

         <class 'pandas.core.frame.DataFrame'>
         RangeIndex: 1000 entries, 0 to 999
         Data columns (total 8 columns):
         First Name           933 non-null object
         Gender               855 non-null object
         Start Date           1000 non-null datetime64[ns]
         Last Login Time      1000 non-null datetime64[ns]
         Salary               1000 non-null int64
         Bonus %              1000 non-null float64
         Senior Management    933 non-null object
         Team                 957 non-null object
         dtypes: datetime64[ns](2), float64(1), int64(1), object(4)
         memory usage: 62.6+ KB
```

Pandas DataFrame

2. Change Gender and Senior Management types

- We have learned about astype() method which is for changing the data type of columns.
- Gender only has 2 unique values. Hence, it is better to change the type to category.
- Notice the memory usage changes from 62.6KB to 49KB. Imagine if we have 1 Million rows. These changes do affect memory performance.

```
In [78]: employees["Gender"] = employees["Gender"].astype("category")

In [79]: employees["Senior Management"] = employees["Senior Management"].astype("bool")

In [80]: employees.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 8 columns):
First Name           933 non-null object
Gender               855 non-null category
Start Date           1000 non-null datetime64[ns]
Last Login Time      1000 non-null datetime64[ns]
Salary               1000 non-null int64
Bonus %              1000 non-null float64
Senior Management    1000 non-null bool
Team                 957 non-null object
dtypes: bool(1), category(1), datetime64[ns](2), float64(1), int64(1), object(2)
memory usage: 49.0+ KB
```

Pandas DataFrame

Filter a DataFrame based on condition

- Let's say we want to filter the data of only male employees. The first step we need to do is **create a list of Boolean values** that corresponds to each row of Gender column.
- Next, we **pass** this **list to the dataFrame.** Once we do that, we will get a filtered dataFrame.
- Here is the list of Male Employees.

```
mask = df["Gender"] == "Male"
mask
```

```
0      True
1      True
2     False
3      True
4      True
5      True
6     False
7     False
8     False
9     False
10    False
11    False
12     True
13     True
```

In [83]:
```
df[mask]
```

Out[83]:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	1993-08-06	2019-08-03 12:42:00	97308	6.945	True	Marketing
1	Thomas	Male	1996-03-31	2019-08-03 06:53:00	61933	4.170	True	NaN
3	Jerry	Male	2005-03-04	2019-08-03 13:00:00	138705	9.340	True	Finance
4	Larry	Male	1998-01-24	2019-08-03 16:47:00	101004	1.389	True	Client Services
5	Dennis	Male	1987-04-18	2019-08-03 01:35:00	115163	10.125	False	Legal
12	Brandon	Male	1980-12-01	2019-08-03 01:08:00	112807	17.492	True	Human Resources

Pandas DataFrame

Filter a DataFrame based on condition

- In this second example, we want to extract **Employees that are in Finance Team.**
- As the first step, we do the same thing which is creating a list of Boolean values by checking every row with a condition.
- Then, we pass the list into Pandas DataFrame.

```
In [84]: extract = df["Team"] == "Finance"
         extract
```

```
Out[84]: 0      False
         1      False
         2       True
         3       True
         4      False
         5      False
         6      False
         7       True
         8      False
         9      False
         10     False
         11     False
```

```
In [85]: df[extract]
```

Out[85]:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
2	Maria	Female	1993-04-23	2019-08-03 11:17:00	130590	11.858	False	Finance
3	Jerry	Male	2005-03-04	2019-08-03 13:00:00	138705	9.340	True	Finance
7	NaN	Female	2015-07-20	2019-08-03 10:43:00	45906	11.598	True	Finance
14	Kimberly	Female	1999-01-14	2019-08-03 07:13:00	41426	14.543	True	Finance
46	Bruce	Male	2009-11-28	2019-08-03 22:47:00	114796	6.796	False	Finance
53	Alan	NaN	2014-03-03	2019-08-03 13:28:00	40341	17.578	True	Finance
56	Carl	Male	2006-05-03	2019-08-03 17:55:00	130276	16.084	True	Finance
67	Rachel	Female	1999-08-16	2019-08-03 06:53:00	51178	9.735	True	Finance

Pandas DataFrame

Filter a DataFrame based on condition

- We can also do it for the **date** column since our Start Time is **datetime data type**. Here, we want to extract employees that have been working since 1990s.

```
In [86]: extract = df["Start Date"] <= "1990-01-01"
         extract
```

```
Out[86]: 0      False
         1      False
         2      False
         3      False
         4      False
         5       True
         6       True
         7      False
         8      False
         9      False
         10      True
         11     False
         12      True
```

```
In [87]: df[extract]
```

Out[87]:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
5	Dennis	Male	1987-04-18	2019-08-03 01:35:00	115163	10.125	False	Legal
6	Ruby	Female	1987-08-17	2019-08-03 16:20:00	65476	10.012	True	Product
10	Louise	Female	1980-08-12	2019-08-03 09:01:00	63241	15.132	True	NaN
12	Brandon	Male	1980-12-01	2019-08-03 01:08:00	112807	17.492	True	Human Resources
17	Shawn	Male	1986-12-07	2019-08-03 19:45:00	111737	6.414	False	Product
18	Diana	Female	1981-10-23	2019-08-03 10:27:00	132940	19.082	False	Client Services
28	Terry	Male	1981-11-27	2019-08-03 18:30:00	124008	13.464	True	Client Services
37	Linda	Female	1981-10-19	2019-08-03 20:49:00	57427	9.557	True	Client Services
38	Stephanie	Female	1986-09-13	2019-08-03 01:52:00	36844	5.574	True	Business Development
43	Marilyn	Female	1980-12-07	2019-08-03 03:16:00	73524	5.207	True	Marketing

Pandas DataFrame

Filtering DataFrame with more than one condition

- If we have two or more conditions when filtering our DataFrame, we need to combine the Boolean lists with **& operators for AND** and **| operators for OR**
- Create **two variables for each condition**. Then, combine these two variables with & symbol and pass it to the DataFrame.

```
In [90]: mask1 = df["Gender"] == "Male"
         mask2 = df["Team"] == "Finance"
         df[mask1 & mask2].head()
```

Out[90]:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
3	Jerry	Male	2005-03-04	2019-08-03 13:00:00	138705	9.340	True	Finance
46	Bruce	Male	2009-11-28	2019-08-03 22:47:00	114796	6.796	False	Finance
56	Carl	Male	2006-05-03	2019-08-03 17:55:00	130276	16.084	True	Finance
68	Jose	Male	2004-10-30	2019-08-03 13:39:00	84834	14.330	True	Finance
83	Shawn	Male	2005-09-23	2019-08-03 02:55:00	148115	6.539	True	Finance

Pandas DataFrame

The isin() method

- **Check** whether each **element in the DataFrame** contains the values we want.
- For example, we want to extract employees who are either in team Finance, Sales or Product. We can use the isin() method as follows.

```
In [94]: mask_isin = df["Team"].isin(["Finance", "Product" , "Sales"])
         mask_isin
```

```
Out[94]: 0     False
         1     False
         2      True
         3      True
         4     False
         5     False
```

```
In [95]: df[mask_isin]
```

Out[95]:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
2	Maria	Female	1993-04-23	2019-08-03 11:17:00	130590	11.858	False	Finance
3	Jerry	Male	2005-03-04	2019-08-03 13:00:00	138705	9.340	True	Finance
6	Ruby	Female	1987-08-17	2019-08-03 16:20:00	65476	10.012	True	Product
7	NaN	Female	2015-07-20	2019-08-03 10:43:00	45906	11.598	True	Finance
13	Gary	Male	2008-01-27	2019-08-03 23:40:00	109831	5.831	False	Sales
14	Kimberly	Female	1999-01-14	2019-08-03 07:13:00	41426	14.543	True	Finance
15	Lillian	Female	2016-06-05	2019-08-03 06:09:00	59414	1.256	False	Product

Pandas DataFrame

The isin() method

- We can also get the same result using == operators.
- However, the code might be longer.

In [93]:
```
mask1 = df["Team"] == "Finance"
mask2 = df["Team"] == "Sales"
mask3 = df["Team"] == "Product"

df[mask1 | mask2 | mask3]
```

Out[93]:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
2	Maria	Female	1993-04-23	2019-08-03 11:17:00	130590	11.858	False	Finance
3	Jerry	Male	2005-03-04	2019-08-03 13:00:00	138705	9.340	True	Finance
6	Ruby	Female	1987-08-17	2019-08-03 16:20:00	65476	10.012	True	Product
7	NaN	Female	2015-07-20	2019-08-03 10:43:00	45906	11.598	True	Finance
13	Gary	Male	2008-01-27	2019-08-03 23:40:00	109831	5.831	False	Sales
14	Kimberly	Female	1999-01-14	2019-08-03 07:13:00	41426	14.543	True	Finance

Pandas DataFrame

isnull() and notnull() methods

- **isnull()** : checks each row in the Series if the element is null/ NaN. Returns a list of Boolean values, True or False.
- **notnull()** : opposite of isnull(). Checks each row if the element exists.

- Filtering rows that have missing Gender values.

```
In [97]: df[df["Gender"].isnull()]
```

Out[97]:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
20	Lois	NaN	1995-04-22	2019-08-03 19:18:00	64714	4.934	True	Legal
22	Joshua	NaN	2012-03-08	2019-08-03 01:58:00	90816	18.816	True	Client Services
27	Scott	NaN	1991-07-11	2019-08-03 18:58:00	122367	5.218	False	Legal
31	Joyce	NaN	2005-02-20	2019-08-03 14:40:00	88657	12.752	False	Product
41	Christine	NaN	2015-06-28	2019-08-03 01:08:00	66582	11.308	True	Business Development
49	Chris	NaN	1980-01-24	2019-08-03 12:13:00	113590	3.055	False	Sales
51	NaN	NaN	2011-12-17	2019-08-03 08:29:00	41126	14.009	True	Sales
53	Alan	NaN	2014-03-03	2019-08-03 13:28:00	40341	17.578	True	Finance
60	Paula	NaN	2005-11-23	2019-08-03 14:01:00	48866	4.271	False	Distribution
64	Kathleen	NaN	1990-04-11	2019-08-03 18:46:00	77834	18.771	False	Business Development
69	Irene	NaN	2015-07-14	2019-08-03 16:31:00	100863	4.382	True	Finance

Pandas DataFrame

isnull() and notnull() methods

- If we only want DataFrame that **does not have** missing values in Gender column, we can use **notnull() method.** This method is used to filter non-missing values.

```
In [98]: df[df["Gender"].notnull()]
```

Out[98]:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	1993-08-06	2019-08-03 12:42:00	97308	6.945	True	Marketing
1	Thomas	Male	1996-03-31	2019-08-03 06:53:00	61933	4.170	True	NaN
2	Maria	Female	1993-04-23	2019-08-03 11:17:00	130590	11.858	False	Finance
3	Jerry	Male	2005-03-04	2019-08-03 13:00:00	138705	9.340	True	Finance
4	Larry	Male	1998-01-24	2019-08-03 16:47:00	101004	1.389	True	Client Services
5	Dennis	Male	1987-04-18	2019-08-03 01:35:00	115163	10.125	False	Legal
6	Ruby	Female	1987-08-17	2019-08-03 16:20:00	65476	10.012	True	Product
7	NaN	Female	2015-07-20	2019-08-03 10:43:00	45906	11.598	True	Finance
8	Angela	Female	2005-11-22	2019-08-03 06:29:00	95570	18.523	True	Engineering

Pandas DataFrame

.between() method

- This function **returns** a Boolean vector containing **True** whenever the corresponding Series element is **between the boundary values left and right**. NA values are treated as False.
- For instance, we want to **filter Salary between 50,000 to 80,000.** Hence, the first variable is the left boundary, and second variable is the right boundary.

```
In [99]: df["Salary"].between(50000,80000)

Out[99]: 0      False
         1       True
         2      False
         3      False
         4      False
         5      False
         6       True
         7      False
         8      False
         9      False
         10      True
```

```
In [100]: df[df["Salary"].between(50000,80000)]

Out[100]:
```

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
1	Thomas	Male	1996-03-31	2019-08-03 06:53:00	61933	4.170	True	NaN
6	Ruby	Female	1987-08-17	2019-08-03 16:20:00	65476	10.012	True	Product
10	Louise	Female	1980-08-12	2019-08-03 09:01:00	63241	15.132	True	NaN
15	Lillian	Female	2016-06-05	2019-08-03 06:09:00	59414	1.256	False	Product
20	Lois	NaN	1995-04-22	2019-08-03 19:18:00	64714	4.934	True	Legal

Pandas DataFrame

.duplicated() method

- Returns Boolean Series denoting duplicate rows, optionally only considering certain columns.
- parameter **keep** :
  - *first* : Mark duplicates as True except for the first occurrence.
  - *last* : Mark duplicates as True except for the last occurrence.
  - *False* : Mark all duplicates as True.

```
In [103]: df["First Name"].duplicated()

Out[103]: 101    False
          327     True
          440     True
          937     True
          137    False
          141     True
          302     True
          538     True
          300    False
          53      True
          610     True
          372    False
          458     True
          477     True
```

```
In [104]: df[df["First Name"].duplicated()].head()
```

Out[104]:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
327	Aaron	Male	1994-01-29	2019-08-03 18:48:00	58755	5.097	True	Marketing
440	Aaron	Male	1990-07-22	2019-08-03 14:53:00	52119	11.343	True	Client Services
937	Aaron	NaN	1986-01-22	2019-08-03 19:39:00	63126	18.424	False	Client Services
141	Adam	Male	1990-12-24	2019-08-03 20:57:00	110194	14.727	True	Product
302	Adam	Male	2007-07-05	2019-08-03 11:59:00	71276	5.027	True	Human Resources

Pandas DataFrame

.duplicated() method

- The difference between keep="first" (default value) and keep="last" is the rows they choose to keep.

```
In [105]: df[df["First Name"].duplicated(keep="last")].head()
```

Out[105]:

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
101	Aaron	Male	2012-02-17	2019-08-03 10:20:00	61602	11.849	True	Marketing
327	Aaron	Male	1994-01-29	2019-08-03 18:48:00	58755	5.097	True	Marketing
440	Aaron	Male	1990-07-22	2019-08-03 14:53:00	52119	11.343	True	Client Services
137	Adam	Male	2011-05-21	2019-08-03 01:45:00	95327	15.120	False	Distribution
141	Adam	Male	1990-12-24	2019-08-03 20:57:00	110194	14.727	True	Product

Pandas DataFrame

.drop_duplicated()

- **Returns** DataFrame with **duplicate rows removed**, optionally only considering certain columns. Indexes, including time indexes, are ignored.
- We have to be specific on the column that we are checking for the duplicates. Else, it will check the entire column to find a row that has the exact same value with the other row.

BEFORE

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
101	Aaron	Male	2012-02-17	2019-08-03 10:20:00	61602	11.849	True	Marketing
327	Aaron	Male	1994-01-29	2019-08-03 18:48:00	58755	5.097	True	Marketing
440	Aaron	Male	1990-07-22	2019-08-03 14:53:00	52119	11.343	True	Client Services
937	Aaron	NaN	1986-01-22	2019-08-03 19:39:00	63126	18.424	False	Client Services
137	Adam	Male	2011-05-21	2019-08-03 01:45:00	95327	15.120	False	Distribution
141	Adam	Male	1990-12-24	2019-08-03 20:57:00	110194	14.727	True	Product
302	Adam	Male	2007-07-05	2019-08-03 11:59:00	71276	5.027	True	Human Resources
538	Adam	Male	2010-10-08	2019-08-03 21:53:00	45181	3.491	False	Human Resources

AFTER

```
df.drop_duplicates(subset=["First Name"], inplace=True, keep=False)
df
```

	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
8	Angela	Female	2005-11-22	2019-08-03 06:29:00	95570	18.523	True	Engineering
688	Brian	Male	2007-04-07	2019-08-03 22:47:00	93901	17.821	True	Legal
190	Carol	Female	1996-03-19	2019-08-03 03:39:00	57783	9.129	False	Finance
887	David	Male	2009-12-05	2019-08-03 08:48:00	92242	15.407	False	Legal
5	Dennis	Male	1987-04-18	2019-08-03 01:35:00	115163	10.125	False	Legal
495	Eugene	Male	1984-05-24	2019-08-03 10:54:00	81077	2.117	False	Sales

Pandas DataFrame

.unique() and .nunique() methods

- unique() method returns unique values of **Series object**.
- Meanwhile, nunique() method return the **number** of unique values
- unique() method **counts** NaN/Null as a unique value.
- nunique() method **does NOT count** NaN/null values.

unique()

```
In [204]: df["Team"].unique()

Out[204]: array(['Marketing', nan, 'Finance', 'Client Services', 'Legal', 'Product',
                 'Engineering', 'Business Development', 'Human Resources', 'Sales',
                 'Distribution'], dtype=object)
```

nunique()

```
In [111]: df["Team"].nunique()

Out[111]: 10
```

```
In [112]: df["Team"].nunique(dropna=False)

Out[112]: 11
```

DATA ANALYSIS WITH PANDAS

Module 3 : DataFrame Part 3

Pandas DataFrame

set_index() and reset_index()

- The Index is automatically generated in numeric.
- However, we can specify if we want a certain **column to be the index, such as Date.**

```
In [114]: bond = pd.read_csv("data/jamesbond.csv")
          bond.head(3)
```

Out[114]:

	Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
0	Dr. No	1962	Sean Connery	Terence Young	448.8	7.0	0.6
1	From Russia with Love	1963	Sean Connery	Terence Young	543.8	12.6	1.6
2	Goldfinger	1964	Sean Connery	Guy Hamilton	820.4	18.6	3.2

```
In [115]: bond.set_index("Film", inplace=True)
          bond.head(3)
```

Out[115]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
Dr. No	1962	Sean Connery	Terence Young	448.8	7.0	0.6
From Russia with Love	1963	Sean Connery	Terence Young	543.8	12.6	1.6
Goldfinger	1964	Sean Connery	Guy Hamilton	820.4	18.6	3.2

Pandas DataFrame

set_index() and reset_index()

- To reset **the index back to numeric** and create a column for the previous index, we can use the reset_index() method.

```
In [115]: bond.set_index("Film", inplace=True)
          bond.head(3)
```

Out[115]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
Dr. No	1962	Sean Connery	Terence Young	448.8	7.0	0.6
From Russia with Love	1963	Sean Connery	Terence Young	543.8	12.6	1.6
Goldfinger	1964	Sean Connery	Guy Hamilton	820.4	18.6	3.2

```
In [116]: bond.reset_index(inplace=True)
          bond.head(3)
```

Out[116]:

	Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
0	Dr. No	1962	Sean Connery	Terence Young	448.8	7.0	0.6
1	From Russia with Love	1963	Sean Connery	Terence Young	543.8	12.6	1.6
2	Goldfinger	1964	Sean Connery	Guy Hamilton	820.4	18.6	3.2

Pandas DataFrame

Retrieve Row(s) by Index Position with **iloc() method**

- iloc means index location.
- Each DataFrame has Index Position Number.
- Extract a **Series** from DataFrame by a specific Index Number.
- Convert **Series into DataFrame** by adding an extra square bracket

```
In [119]: bond = pd.read_csv("data/jamesbond.csv")
          bond.sort_index(inplace=True)
          bond.head(3)
```

Out[119]:

	Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
0	Dr. No	1962	Sean Connery	Terence Young	448.8	7.0	0.6
1	From Russia with Love	1963	Sean Connery	Terence Young	543.8	12.6	1.6
2	Goldfinger	1964	Sean Connery	Guy Hamilton	820.4	18.6	3.2

```
In [206]: bond.iloc[1]
```

```
Out[206]: Film                 From Russia with Love
          Year                                 1963
          Actor                       Sean Connery
          Director                  Terence Young
          Box Office                          543.8
          Budget                               12.6
          Bond Actor Salary                     1.6
          Name: 1, dtype: object
```

```
In [207]: bond.iloc[[1]]
```

Out[207]:

	Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
1	From Russia with Love	1963	Sean Connery	Terence Young	543.8	12.6	1.6

Pandas DataFrame

Retrieve Row(s) by Index Position with **iloc() method**

- Extract a specific position using a **list of indexes.**

```
In [122]: bond.iloc[[5,10,15,20]]
```

Out[122]:

	Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
5	You Only Live Twice	1967	Sean Connery	Lewis Gilbert	514.2	59.9	4.4
10	The Spy Who Loved Me	1977	Roger Moore	Lewis Gilbert	533.0	45.1	NaN
15	A View to a Kill	1985	Roger Moore	John Glen	275.2	54.5	9.1
20	The World Is Not Enough	1999	Pierce Brosnan	Michael Apted	439.5	158.3	13.5

- A **range of rows** can be extracted by separating the upper bound and lower bound with **colon (:)**

```
In [123]: bond.iloc[3:5]
```

Out[123]:

	Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
3	Thunderball	1965	Sean Connery	Terence Young	848.1	41.9	4.7
4	Casino Royale	1967	David Niven	Ken Hughes	315.0	85.0	NaN

Pandas DataFrame

Retrieve row by Index Label using **.loc[ ] method**

- Access a group of rows and columns by label(s) or a boolean array.
- We know that we can specify names as the index. If our index is string labelled, we have to write the **index label inside the square bracket**. It will **return a Series.**

```
In [124]: bond = pd.read_csv("data/jamesbond.csv", index_col="Film")
          bond.sort_index(inplace=True)
          bond.head(6)
```

Out[124]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
A View to a Kill	1985	Roger Moore	John Glen	275.2	54.5	9.1
Casino Royale	2006	Daniel Craig	Martin Campbell	581.5	145.3	3.3
Casino Royale	1967	David Niven	Ken Hughes	315.0	85.0	NaN
Diamonds Are Forever	1971	Sean Connery	Guy Hamilton	442.5	34.7	5.8
Die Another Day	2002	Pierce Brosnan	Lee Tamahori	465.4	154.2	17.9
Dr. No	1962	Sean Connery	Terence Young	448.8	7.0	0.6

```
In [126]: bond.loc[["Die Another Day"]]
```

Out[126]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
Die Another Day	2002	Pierce Brosnan	Lee Tamahori	465.4	154.2	17.9

```
In [125]: bond.loc["Die Another Day"]
```

Out[125]:

```
Year                  2002
Actor          Pierce Brosnan
Director         Lee Tamahori
Box Office              465.4
Budget                  154.2
Bond Actor Salary        17.9
Name: Die Another Day, dtype: object
```

Pandas DataFrame

Retrieve row by Index Label using **.loc[ ] method**

- Next, let's access a **range of rows**. Specify two index labels and separate them with a colon (:)

```
In [124]: bond = pd.read_csv("data/jamesbond.csv", index_col="Film")
          bond.sort_index(inplace=True)
          bond.head(6)
```

Out[124]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
A View to a Kill	1985	Roger Moore	John Glen	275.2	54.5	9.1
Casino Royale	2006	Daniel Craig	Martin Campbell	581.5	145.3	3.3
Casino Royale	1967	David Niven	Ken Hughes	315.0	85.0	NaN
Diamonds Are Forever	1971	Sean Connery	Guy Hamilton	442.5	34.7	5.8
Die Another Day	2002	Pierce Brosnan	Lee Tamahori	465.4	154.2	17.9
Dr. No	1962	Sean Connery	Terence Young	448.8	7.0	0.6

```
In [127]: bond.loc["Casino Royale" : "Die Another Day"]
```

Out[127]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
Casino Royale	2006	Daniel Craig	Martin Campbell	581.5	145.3	3.3
Casino Royale	1967	David Niven	Ken Hughes	315.0	85.0	NaN
Diamonds Are Forever	1971	Sean Connery	Guy Hamilton	442.5	34.7	5.8
Die Another Day	2002	Pierce Brosnan	Lee Tamahori	465.4	154.2	17.9

Pandas DataFrame

Specific only two rows

- Put the index label in a square bracket separated by a comma (,)

```
In [128]: bond.loc[["Diamonds Are Forever" ,"A View to a Kill" ]]
```

Out[128]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
Diamonds Are Forever	1971	Sean Connery	Guy Hamilton	442.5	34.7	5.8
A View to a Kill	1985	Roger Moore	John Glen	275.2	54.5	9.1

- iloc() can also be used even though the index is String labelled.

```
In [129]: bond.iloc[[0]]
```

Out[129]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
A View to a Kill	1985	Roger Moore	John Glen	275.2	54.5	9.1

Pandas DataFrame

.ix[ ] method

- Works the same as .loc[ ] and iloc[ ]. However, it is **deprecated** and advised to use loc[ ] or iloc[ ]

```
In [131]: bond.ix[["A View to a Kill"]]
```

Out[131]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
A View to a Kill	1985	Roger Moore	John Glen	275.2	54.5	9.1

Pandas DataFrame

Second Arguments in .loc[ ] and iloc[ ]

- The second argument indicates the column that we need.
- Combining the first and second argument enables us to extract very specific data.
- Specific data of *"A View to a Kill "* movie. E.g: the budget
  - **loc** : the arguments can only be string typed
  - **iloc** : the arguments can only be integer typed
  - **ix** : Arguments can be both integer and string. However, this method is deprecated.

```
In [210]: bond.loc["A View to a Kill"]

Out[210]: Year                        1985
          Actor                Roger Moore
          Director               John Glen
          Box Office                 275.2
          Budget                      54.5
          Bond Actor Salary            9.1
          Name: A View to a Kill, dtype: object
```

```
In [134]: bond.loc["A View to a Kill" , "Budget"]

Out[134]: 54.5
```

```
In [135]: bond.iloc[0 , 4]

Out[135]: 54.5
```

Pandas DataFrame

Set a New Values for a Specific Cell or Row

- Firstly, we extract the data then reassign it.
- If we add equal sign (=) to a value, the cell will be replaced with the new value.

```
In [138]: bond.loc["A View to a Kill", "Year"]
Out[138]: 1985
```

```
In [139]: bond.loc["A View to a Kill", "Year"] = 2019
          bond.head(3)
```

Out[139]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
A View to a Kill	2019	Roger Moore	John Glen	275.2	54.5	9.1
Casino Royale	2006	Daniel Craig	Martin Campbell	581.5	145.3	3.3
Casino Royale	1967	David Niven	Ken Hughes	315.0	85.0	NaN

Pandas DataFrame

Set a New Values for a Specific Cell or Row

- We can also **change more than one cell** by calling the list of columns and then assigning them with the new values as follows.

```
In [140]: bond.loc["A View to a Kill" , ["Year", "Budget" , "Actor"]]

Out[140]: Year             2019
          Budget           54.5
          Actor     Roger Moore
          Name: A View to a Kill, dtype: object
```

```
In [141]: bond.loc["A View to a Kill" , ["Year", "Budget" , "Actor"]] = [2020, 50, "Ali bin Abu" ]
          bond.head(3)
```

Out[141]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
A View to a Kill	2020	Ali bin Abu	John Glen	275.2	50.0	9.1
Casino Royale	2006	Daniel Craig	Martin Campbell	581.5	145.3	3.3
Casino Royale	1967	David Niven	Ken Hughes	315.0	85.0	NaN

Pandas DataFrame

Set Multiple Values in DataFrame

- For example, we want to change the Actor's name, Roger Moore to Amin Hakim. To do that, we need to extract the specific rows first.

```
In [143]: mask = bond.Actor == "Roger Moore"
          bond.loc[mask]
```

Out[143]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
A View to a Kill	1985	Roger Moore	John Glen	275.2	54.5	9.1
For Your Eyes Only	1981	Roger Moore	John Glen	449.4	60.2	NaN
Live and Let Die	1973	Roger Moore	Guy Hamilton	460.3	30.8	NaN
Moonraker	1979	Roger Moore	Lewis Gilbert	535.0	91.5	NaN
Octopussy	1983	Roger Moore	John Glen	373.8	53.9	7.8

- Then, we set the second argument as the name of the column that we want to change. Lastly, we assign the new value by using the equal (=) sign.

```
In [144]: bond.loc[mask, "Actor"] = "Amin Hakim"
          bond.loc[mask]
```

Out[144]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
A View to a Kill	1985	Amin Hakim	John Glen	275.2	54.5	9.1
For Your Eyes Only	1981	Amin Hakim	John Glen	449.4	60.2	NaN
Live and Let Die	1973	Amin Hakim	Guy Hamilton	460.3	30.8	NaN
Moonraker	1979	Amin Hakim	Lewis Gilbert	535.0	91.5	NaN
Octopussy	1983	Amin Hakim	John Glen	373.8	53.9	7.8

Pandas DataFrame

Rename Index Labels or Columns in DataFrame

- The column parameter receives a dictionary.
- The **dictionary key** is **the column name** that we want to replace. Meanwhile, the **value** will be the **new name for the column.**

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
Dr. No	1962	Sean Connery	Terence Young	448.8	7.0	0.6
From Russia with Love	1963	Sean Connery	Terence Young	543.8	12.6	1.6
Goldfinger	1964	Sean Connery	Guy Hamilton	820.4	18.6	3.2

```
In [146]: bond.rename(columns= {
              "Year" : "Released Date",
              "Box Office" : "Revenue"
          }, inplace=True)

In [147]: bond.head()
```

Out[147]:

Film	Released Date	Actor	Director	Revenue	Budget	Bond Actor Salary
Dr. No	1962	Sean Connery	Terence Young	448.8	7.0	0.6
From Russia with Love	1963	Sean Connery	Terence Young	543.8	12.6	1.6
Goldfinger	1964	Sean Connery	Guy Hamilton	820.4	18.6	3.2
Thunderball	1965	Sean Connery	Terence Young	848.1	41.9	4.7
Casino Royale	1967	David Niven	Ken Hughes	315.0	85.0	NaN

Pandas DataFrame

Rename Index Labels or Columns in DataFrame

- The index parameter receives a dictionary.
- The dictionary key is the index label that we want to replace while the value will be the new value.

Film	Released Date	Actor	Director	Revenue	Budget	Bond Actor Salary
Dr. No	1962	Sean Connery	Terence Young	448.8	7.0	0.6
From Russia with Love	1963	Sean Connery	Terence Young	543.8	12.6	1.6
Goldfinger	1964	Sean Connery	Guy Hamilton	820.4	18.6	3.2
Thunderball	1965	Sean Connery	Terence Young	848.1	41.9	4.7
Casino Royale	1967	David Niven	Ken Hughes	315.0	85.0	NaN

```
In [148]: bond.rename(index={
              "Dr. No" : "Doctor No"
          }, inplace=True)

In [149]: bond.head()
```

Out[149]:

Film	Released Date	Actor	Director	Revenue	Budget	Bond Actor Salary
Doctor No	1962	Sean Connery	Terence Young	448.8	7.0	0.6
From Russia with Love	1963	Sean Connery	Terence Young	543.8	12.6	1.6
Goldfinger	1964	Sean Connery	Guy Hamilton	820.4	18.6	3.2

Pandas DataFrame

Delete Rows or Columns from a DataFrame

- Dropping only a row :

```
In [151]: bond.drop("A View to a Kill", inplace=True)

In [152]: bond.head()
```

Out[152]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
Casino Royale	2006	Daniel Craig	Martin Campbell	581.5	145.3	3.3
Casino Royale	1967	David Niven	Ken Hughes	315.0	85.0	NaN

- Dropping multiple rows :

```
In [153]: bond.drop(["Casino Royale" , "Dr. No"], inplace=True)
          bond.head()
```

Out[153]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
Diamonds Are Forever	1971	Sean Connery	Guy Hamilton	442.5	34.7	5.8
Die Another Day	2002	Pierce Brosnan	Lee Tamahori	465.4	154.2	17.9
For Your Eyes Only	1981	Roger Moore	John Glen	449.4	60.2	NaN

Pandas DataFrame

Delete Rows or Columns from a DataFrame

- When dropping columns, we need to include **columns parameter**. For example, we want to drop the Box Office column.

```
In [154]: bond.drop(columns = "Box Office", inplace=True)
          bond.head()
```

Out[154]:

Film	Year	Actor	Director	Budget	Bond Actor Salary
Diamonds Are Forever	1971	Sean Connery	Guy Hamilton	34.7	5.8
Die Another Day	2002	Pierce Brosnan	Lee Tamahori	154.2	17.9
For Your Eyes Only	1981	Roger Moore	John Glen	60.2	NaN
From Russia with Love	1963	Sean Connery	Terence Young	12.6	1.6
GoldenEye	1995	Pierce Brosnan	Martin Campbell	76.9	5.1

Pandas DataFrame

Extract Random Sample

- Random sampling is a method to choose random sample of items to ensure non-bias selection.
- **n** parameter indicates how many rows we want.

```
In [159]: bond.sample(n = 10)
```

Out[159]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
Casino Royale	2006	Daniel Craig	Martin Campbell	581.5	145.3	3.3
Die Another Day	2002	Pierce Brosnan	Lee Tamahori	465.4	154.2	17.9
The Living Daylights	1987	Timothy Dalton	John Glen	313.5	68.8	5.2
Skyfall	2012	Daniel Craig	Sam Mendes	943.5	170.2	14.5
Live and Let Die	1973	Roger Moore	Guy Hamilton	460.3	30.8	NaN
The Man with the Golden Gun	1974	Roger Moore	Guy Hamilton	334.0	27.7	NaN
Octopussy	1983	Roger Moore	John Glen	373.8	53.9	7.8
Moonraker	1979	Roger Moore	Lewis Gilbert	535.0	91.5	NaN
Tomorrow Never Dies	1997	Pierce Brosnan	Roger Spottiswoode	463.2	133.9	10.0
Thunderball	1965	Sean Connery	Terence Young	848.1	41.9	4.7

Pandas DataFrame

Extract Random Sample

- **Frac** parameter indicates fraction number of rows we want out of the total.
- 0.25 means 25% of the total number of rows. We have a total of 26 rows so 25% will be equal to 6 rows.

```
In [160]: bond.sample(frac = 0.25)
```

Out[160]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
Octopussy	1983	Roger Moore	John Glen	373.8	53.9	7.8
Quantum of Solace	2008	Daniel Craig	Marc Forster	514.2	181.4	8.1
Skyfall	2012	Daniel Craig	Sam Mendes	943.5	170.2	14.5
Tomorrow Never Dies	1997	Pierce Brosnan	Roger Spottiswoode	463.2	133.9	10.0
From Russia with Love	1963	Sean Connery	Terence Young	543.8	12.6	1.6
Goldfinger	1964	Sean Connery	Guy Hamilton	820.4	18.6	3.2

Pandas DataFrame

Extract Random Sample

- We can also extract samples from random column.
- Try re-running the following code. Notice there are ONLY 3 random columns selected.
- axis = 1 OR axis =“columns” means we want the sampling to be done on random column.

```
In [161]: bond.sample(n = 3, axis = 1)
```

Out[161]:

Film	Budget	Actor	Box Office
A View to a Kill	54.5	Roger Moore	275.2
Casino Royale	145.3	Daniel Craig	581.5
Casino Royale	85.0	David Niven	315.0
Diamonds Are Forever	34.7	Sean Connery	442.5
Die Another Day	154.2	Pierce Brosnan	465.4
Dr. No	7.0	Sean Connery	448.8
For Your Eyes Only	60.2	Roger Moore	449.4
From Russia with Love	12.6	Sean Connery	543.8
GoldenEye	76.9	Pierce Brosnan	518.5

Pandas DataFrame

The .query() Method

- Query the columns of a DataFrame using Boolean expressions.
- To use the query method, we first have to replace the spaces in all column names with underscore (_).
- This is because the query method only works if the column name doesn't have any empty spaces.

```
In [170]: bond.columns = [column_name.replace(" ", "_") for column_name in bond.columns]
          bond.head()
```

Out[170]:

Film	Year	Actor	Director	Box_Office	Budget	Bond_Actor_Salary
A View to a Kill	1985	Roger Moore	John Glen	275.2	54.5	9.1
Casino Royale	2006	Daniel Craig	Martin Campbell	581.5	145.3	3.3
Casino Royale	1967	David Niven	Ken Hughes	315.0	85.0	NaN

```
In [171]: bond.query("Actor == 'David Niven'")
```

Out[171]:

Film	Year	Actor	Director	Box_Office	Budget	Bond_Actor_Salary
Casino Royale	1967	David Niven	Ken Hughes	315.0	85.0	NaN

Pandas DataFrame

The .query() Method

- Another example using the query() method.

```
In [172]: bond.query("Box_Office > 500 ")
```

Out[172]:

Film	Year	Actor	Director	Box_Office	Budget	Bond_Actor_Salary
Casino Royale	2006	Daniel Craig	Martin Campbell	581.5	145.3	3.3
From Russia with Love	1963	Sean Connery	Terence Young	543.8	12.6	1.6
GoldenEye	1995	Pierce Brosnan	Martin Campbell	518.5	76.9	5.1
Goldfinger	1964	Sean Connery	Guy Hamilton	820.4	18.6	3.2
Moonraker	1979	Roger Moore	Lewis Gilbert	535.0	91.5	NaN

- Two query conditions

```
In [173]: bond.query("Box_Office > 500 and Actor == 'Daniel Craig'")
```

Out[173]:

Film	Year	Actor	Director	Box_Office	Budget	Bond_Actor_Salary
Casino Royale	2006	Daniel Craig	Martin Campbell	581.5	145.3	3.3
Quantum of Solace	2008	Daniel Craig	Marc Forster	514.2	181.4	8.1
Skyfall	2012	Daniel Craig	Sam Mendes	943.5	170.2	14.5
Spectre	2015	Daniel Craig	Sam Mendes	726.7	206.3	NaN

Pandas DataFrame

Apply() method on Single Column

- **apply()** method enable us to **apply function** along an axis of the DataFrame.
- For example, we want **to add "Millions"** at each cell in the Box Office column. Firstly, we create add_million() function and then use its name as the argument in the apply() method.

Out[174]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
A View to a Kill	1985	Roger Moore	John Glen	275.2	54.5	9.1
Casino Royale	2006	Daniel Craig	Martin Campbell	581.5	145.3	3.3
Casino Royale	1967	David Niven	Ken Hughes	315.0	85.0	NaN

```
In [175]: def add_million(number):
              return " " + str(number) + "Millions"

In [176]: bond["Box Office"] = bond["Box Office"].apply(add_million)

In [177]: bond.head(3)
```

Out[177]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
A View to a Kill	1985	Roger Moore	John Glen	275.2Millions	54.5	9.1
Casino Royale	2006	Daniel Craig	Martin Campbell	581.5Millions	145.3	3.3
Casino Royale	1967	David Niven	Ken Hughes	315.0Millions	85.0	NaN

Pandas DataFrame

.copy() method

- Makes a copy of an object and any changes made on the new object will not affect the original dataset.
- **Preserve the original dataset** (See next slide and compare the output).

```
In [179]: bond = pd.read_csv("data/jamesbond.csv", index_col="Film")
          bond.sort_index(inplace=True)
          bond.head(3)
```

Out[179]:

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
A View to a Kill	1985	Roger Moore	John Glen	275.2	54.5	9.1
Casino Royale	2006	Daniel Craig	Martin Campbell	581.5	145.3	3.3
Casino Royale	1967	David Niven	Ken Hughes	315.0	85.0	NaN

Pandas DataFrame

.copy() method

WITHOUT .COPY()

```
In [228]: directors = bond["Director"]
          directors.head(3)
```

```
Out[228]: Film
          A View to a Kill          John Glen
          Casino Royale       Martin Campbell
          Casino Royale            Ken Hughes
          Name: Director, dtype: object
```

```
In [229]: directors["A View to a Kill"] = "Ali bin Abu"
          directors.head(3)
```

```
Out[229]: Film
          A View to a Kill        Ali bin Abu
          Casino Royale       Martin Campbell
          Casino Royale            Ken Hughes
          Name: Director, dtype: object
```

```
bond.head(3)
```

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
A View to a Kill	1985	Roger Moore	Ali bin Abu	275.2	54.5	9.1
Casino Royale	2006	Daniel Craig	Martin Campbell	581.5	145.3	3.3

USING .COPY()

```
In [225]: directors = bond["Director"].copy()
          directors.head(3)
```

```
Out[225]: Film
          A View to a Kill          John Glen
          Casino Royale       Martin Campbell
          Casino Royale            Ken Hughes
          Name: Director, dtype: object
```

```
In [226]: directors["A View to a Kill"] = "Ali bin Abu"
          directors.head(3)
```

```
Out[226]: Film
          A View to a Kill        Ali bin Abu
          Casino Royale       Martin Campbell
          Casino Royale            Ken Hughes
          Name: Director, dtype: object
```

```
bond.head(3)
```

Film	Year	Actor	Director	Box Office	Budget	Bond Actor Salary
A View to a Kill	1985	Roger Moore	John Glen	275.2	54.5	9.1
Casino Royale	2006	Daniel Craig	Martin Campbell	581.5	145.3	3.3
Casino Royale	1967	David Niven	Ken Hughes	315.0	85.0	NaN

DATA ANALYSIS WITH PYTHON

Module 4: Working with Text Data

Working with Text Data

Introduction

- Preprocessing our data is very important.
- We may encounter data that is not clean. As we can see in DataFrame below:
  - First name and last name are separated by a comma.
  - Employee Annual Salary has a dollar ($) sign, which means we cannot do mathematical operation on it.

In [169]:
```
import pandas as pd
df = pd.read_csv("data/chicago.csv")
df.head()
```

Out[169]:

	Name	Position Title	Department	Employee Annual Salary
0	AARON, ELVIA J	WATER RATE TAKER	WATER MGMNT	$90744.00
1	AARON, JEFFERY M	POLICE OFFICER	POLICE	$84450.00
2	AARON, KARINA	POLICE OFFICER	POLICE	$84450.00
3	AARON, KIMBERLEI R	CHIEF CONTRACT EXPEDITER	GENERAL SERVICES	$89880.00
4	ABAD JR, VICENTE M	CIVIL ENGINEER IV	WATER MGMNT	$106836.00

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32063 entries, 0 to 32062
Data columns (total 4 columns):
Name                        32062 non-null object
Position Title              32062 non-null object
Department                  32062 non-null object
Employee Annual Salary      32062 non-null object
dtypes: object(4)
memory usage: 1002.0+ KB
```

Working with Text Data

Introduction

- In Department column, we have 35 categories. It is much better to change the column type to **category** as we have 32,063 rows of data. Rather than having 32,063 strings, we can classify them into 35 categories.
- You can see that the memory usage decreases from 1002KB to 784.4KB. This may speed up our operation so much more.

```
In [171]: df["Department"].nunique()
Out[171]: 35
```

```
In [172]: df["Department"] = df["Department"].astype("category")

In [173]: df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32063 entries, 0 to 32062
Data columns (total 4 columns):
Name                    32062 non-null object
Position Title          32062 non-null object
Department              32062 non-null category
Employee Annual Salary  32062 non-null object
dtypes: category(1), object(3)
memory usage: 784.4+ KB
```

Working with Text Data

Remove null values

In [174]:
```
df.tail()
```

Out[174]:

	Name	Position Title	Department	Employee Annual Salary
32058	ZYGOWICZ, PETER J	POLICE OFFICER	POLICE	$87384.00
32059	ZYMANTAS, MARK E	POLICE OFFICER	POLICE	$84450.00
32060	ZYRKOWSKI, CARLO E	POLICE OFFICER	POLICE	$87384.00
32061	ZYSKOWSKI, DARIUSZ	CHIEF DATA BASE ANALYST	DoIT	$113664.00
32062	NaN	NaN	NaN	NaN

In [175]:
```
df.dropna(how="all", inplace=True)
df.tail()
```

Out[175]:

	Name	Position Title	Department	Employee Annual Salary
32057	ZYGADLO, MICHAEL J	FRM OF MACHINISTS - AUTOMOTIVE	GENERAL SERVICES	$99528.00
32058	ZYGOWICZ, PETER J	POLICE OFFICER	POLICE	$87384.00
32059	ZYMANTAS, MARK E	POLICE OFFICER	POLICE	$84450.00
32060	ZYRKOWSKI, CARLO E	POLICE OFFICER	POLICE	$87384.00
32061	ZYSKOWSKI, DARIUSZ	CHIEF DATA BASE ANALYST	DoIT	$113664.00

Working with Text Data

Python Built-in function

- **Python built-in function : lower(), upper(), title() , len()**
- **lower() :** returns the string in lowercase.
- **upper() :** returns a copy of the string in uppercase.
- **title() :** convert the first character in each word to uppercase and remaining characters to lowercase in string
- **len() :** calculate the length of string/ list/ etc. (In string, space also counts as a character)

```
In [176]: "HELLO WORLD".lower()
Out[176]: 'hello world'

In [177]: "hello world".upper()
Out[177]: 'HELLO WORLD'

In [178]: "hello world".title()
Out[178]: 'Hello World'

In [179]: len("HELLO WORLD")
Out[179]: 11
```

Working with Text Data

Python Built-in function

- These functions allow us to change the format in Name column into something more readable.

```
In [180]: df["Name"] = df["Name"].str.title()
          df.head()
```

Out[180]:

	Name	Position Title	Department	Employee Annual Salary
0	Aaron, Elvia J	WATER RATE TAKER	WATER MGMNT	$90744.00
1	Aaron, Jeffery M	POLICE OFFICER	POLICE	$84450.00
2	Aaron, Karina	POLICE OFFICER	POLICE	$84450.00
3	Aaron, Kimberlei R	CHIEF CONTRACT EXPEDITER	GENERAL SERVICES	$89880.00
4	Abad Jr, Vicente M	CIVIL ENGINEER IV	WATER MGMNT	$106836.00

- The same goes for Position Title column and Department column.

```
In [181]: df["Position Title"] = df["Position Title"].str.title()
          df.head()
```

Out[181]:

	Name	Position Title	Department	Employee Annual Salary
0	Aaron, Elvia J	Water Rate Taker	WATER MGMNT	$90744.00
1	Aaron, Jeffery M	Police Officer	POLICE	$84450.00
2	Aaron, Karina	Police Officer	POLICE	$84450.00
3	Aaron, Kimberlei R	Chief Contract Expediter	GENERAL SERVICES	$89880.00
4	Abad Jr, Vicente M	Civil Engineer Iv	WATER MGMNT	$106836.00

```
In [182]: df["Department"] = df["Department"].str.title()
          df.head()
```

Out[182]:

	Name	Position Title	Department	Employee Annual Salary
0	Aaron, Elvia J	Water Rate Taker	Water Mgmnt	$90744.00
1	Aaron, Jeffery M	Police Officer	Police	$84450.00
2	Aaron, Karina	Police Officer	Police	$84450.00
3	Aaron, Kimberlei R	Chief Contract Expediter	General Services	$89880.00
4	Abad Jr, Vicente M	Civil Engineer Iv	Water Mgmnt	$106836.00

Working with Text Data

.replace() method

- The function is used to replace a string, regex, list, dictionary, series, or number.
- In the Department column, "Management“ is shortened to "Mgmnt“ while in the Employee Annual Salary column, there is dollar sign ($) in each values. These data can be cleaned using **replace() method.**
- Firstly, we have to understand how it works. The first argument is the pattern that we want to replace. The second argument is what we want it to be replaced with.

```
In [183]: "Hello World".replace("l", "!")

Out[183]: 'He!!o Wor!d'
```

```
In [184]: df["Department"] = df["Department"].str.replace("Mgmnt", "Management")
          df.head(3)
```

Out[184]:

	Name	Position Title	Department	Employee Annual Salary
0	Aaron, Elvia J	Water Rate Taker	Water Management	$90744.00
1	Aaron, Jeffery M	Police Officer	Police	$84450.00
2	Aaron, Karina	Police Officer	Police	$84450.00

Working with Text Data

.replace() method

- Let's remove the dollar sign in Employee Annual Salary column and change the data type to float.

```
In [185]: df["Employee Annual Salary"] = df["Employee Annual Salary"].str.replace("$", "").astype(float)
          df.head(3)
```

Out[185]:

	Name	Position Title	Department	Employee Annual Salary
0	Aaron, Elvia J	Water Rate Taker	Water Management	90744.0
1	Aaron, Jeffery M	Police Officer	Police	84450.0
2	Aaron, Karina	Police Officer	Police	84450.0

- Now, we can do mathematical operations on that column. We can get the average annual salary which is 80,204.

```
In [186]: df["Employee Annual Salary"].mean()
Out[186]: 80204.178633899
```

Working with Text Data

.contains() , endswith() startswith() methods

- These methods return Boolean values, either True or False
- Let see how many jobs that are related to water. Out of 32,061, only 111 jobs are related to water.

```
In [187]: mask = df["Position Title"].str.contains("Water")
          df[mask]
```

Out[187]:

	Name	Position Title	Department	Employee Annual Salary
0	Aaron, Elvia J	Water Rate Taker	Water Management	90744.0
554	Aluise, Vincent G	Foreman Of Water Pipe Construction	Water Management	102440.0
671	Ander, Perry A	Water Chemist Ii	Water Management	82044.0
685	Anderson, Andrew J	District Superintendent Of Water Distribution	Water Management	109272.0
702	Anderson, Donald	Foreman Of Water Pipe Construction	Water Management	102440.0
1054	Ashley, Karma T	Water Chemist Ii	Water Management	82044.0
1079	Atkins, Joanna M	Water Chemist Ii	Water Management	82044.0
1181	Azeem, Mohammed A	Water Chemist Ii	Water Management	53172.0
1285	Bajic, John A	Water Meter Machinist	Water Management	82576.0
2400	Bolton, Brian E	Water Rate Taker	Water Management	78948.0
2586	Boyce, Adner L	Water Chemist Ii	Water Management	82044.0
2745	Brandys, Daniel	Water Chemist Ii	Water Management	53172.0
3143	Brown, Sharon L	Water Rate Taker	Water Management	82728.0
3246	Buchanan, Anthony L	Water Chemist Ii	Water Management	66780.0
3456	Burt, Carl S	Water Rate Taker	Water Management	90744.0

Working with Text Data

.contains() , endswith() startswith() methods

- We can **identify specialists** by using **endswith() method.** Here, we have 100 specialists from different departments

```
In [188]: mask = df["Position Title"].str.endswith("Specialist")
          df[mask]
```

Out[188]:

	Name	Position Title	Department	Employee Annual Salary
308	Alarcon, Luis J	Loan Processing Specialist	Community Development	81948.00
422	Allain, Carolyn	Senior Telecommunications Specialist	Doit	89880.00
705	Anderson, Edward M	Sr Procurement Specialist	Procurement	91476.00
1163	Ayala Jr, Juan	Field Sanitation Specialist	Streets & San	78948.00
1558	Barrett, Barbara J	Technical Training Specialist	Police	94200.00
1869	Beltran, Mauricio	Procurement Specialist	Procurement	79596.00
2095	Biggane, Thomas M	Security Specialist	City Council	52000.08
2319	Bobba, Mahita	Grants Research Specialist	Health	84924.00
2519	Boston, Nathaniel K	Field Sanitation Specialist	Streets & San	75384.00
2678	Bradley, Jeena	Grants Research Specialist	Health	97812.00
4109	Carter, Alice	Safety Specialist	Water Management	86580.00
4479	Chan, Joseph	Sr Procurement Specialist	Procurement	83340.00
5043	Coco, Michelle T	Investigator Specialist	Fire	91476.00
5307	Condon, Kristin	Hr Records Specialist	Human Resources	48852.00
5633	Cortes, Jezieel T	Sr Procurement Specialist	Procurement	68556.00

Working with Text Data

.contains() , endswith() startswith() methods

- We can also do **identification** by using **startswith() method.**

```
In [189]: mask = df["Position Title"].str.startswith("Water")
          df[mask]
```

Out[189]:

	Name	Position Title	Department	Employee Annual Salary
0	Aaron, Elvia J	Water Rate Taker	Water Management	90744.0
671	Ander, Perry A	Water Chemist Ii	Water Management	82044.0
1054	Ashley, Karma T	Water Chemist Ii	Water Management	82044.0
1079	Atkins, Joanna M	Water Chemist Ii	Water Management	82044.0
1181	Azeem, Mohammed A	Water Chemist Ii	Water Management	53172.0
1285	Bajic, John A	Water Meter Machinist	Water Management	82576.0
2400	Bolton, Brian E	Water Rate Taker	Water Management	78948.0
2586	Boyce, Adner L	Water Chemist Ii	Water Management	82044.0
2745	Brandys, Daniel	Water Chemist Ii	Water Management	53172.0
3143	Brown, Sharon L	Water Rate Taker	Water Management	82728.0
3246	Buchanan, Anthony L	Water Chemist Ii	Water Management	66780.0
3456	Burt, Carl S	Water Rate Taker	Water Management	90744.0
3626	Cage, Ophelia	Water Rate Taker	Water Management	90744.0
4889	Clark, Mark A	Water Meter Machinist	Water Management	82576.0

Working with Text Data

strip() , lstrip() , rstrip()

- Strip means removing white space at the start or end of a string.
- **lstrip()** stands for **left** strip. It will only remove white space on the left. On the other hand, **rstrip()** will only remove white space on the **right.**
- These methods are very handy to clean unneeded spaces in the data. Sometimes users tend to make mistakes on input.

```
In [190]: "         Hello Word    ".strip()
Out[190]: 'Hello Word'

In [191]: "         Hello Word    ".lstrip()
Out[191]: 'Hello Word    '

In [192]: "         Hello Word    ".rstrip()
Out[192]: '         Hello Word'

In [193]: df["Name"] = df["Name"].str.strip()
          df["Name"].head()
Out[193]: 0        Aaron,  Elvia J
          1      Aaron,  Jeffery M
          2         Aaron,  Karina
          3    Aaron,  Kimberlei R
          4    Abad Jr,  Vicente M
          Name: Name, dtype: object
```

Working with Text Data

.str.split() method

- Split string by a character using .str.split() method
- It will split strings around given separator/delimiter.
- The split() method will divide the string based on the separator that we specify. In the example, the separator is a space (" ")
- We can separate the Last name and first name in Name column.

```
In [194]: "My name is Ali bin Abu".split(" ")
Out[194]: ['My', 'name', 'is', 'Ali', 'bin', 'Abu']
```

```
In [164]: lastName =  df["Name"].str.split(",").str.get(0).str.strip()
          lastName
Out[164]: 0          Aaron
          1          Aaron
          2          Aaron
          3          Aaron
          4       Abad Jr
          5        Abarca
```

```
In [196]: firstName =  df["Name"].str.split(",").str.get(1).str.strip().str.split(" ").str.get(0)
          firstName
Out[196]: 0          Elvia
          1        Jeffery
          2         Karina
          3       Kimberlei
          4         Vicente
          5          Anabel
          6        Emmanuel
          7           Reece
          8      Christopher
          9           Robert
          10          James
          11          Terry
          12          Betty
          13         Lynise
          14        William
          15           Zaid
          16      Abdalmahd
```

Working with Text Data

.str.split() method

- We can separate the Last name and first name in the Name column.

```
firstName =  df["Name"].str.split(",").str.get(1).str.strip().str.split(" ").str.get(0)
firstName
```

```
0             Elvia
1           Jeffery
2            Karina
3         Kimberlei
4           Vicente
5            Anabel
6          Emmanuel
7             Reece
8       Christopher
9            Robert
10            James
11            Terry
12            Betty
13           Lynise
14          William
15             Zaid
16       Abdalmahd
```

```
lastName =  df["Name"].str.split(",").str.get(0).str.strip()
lastName
```

```
0              Aaron
1              Aaron
2              Aaron
3              Aaron
4            Abad Jr
5             Abarca
6             Abarca
7            Abascal
8              Abbasi
9           Abbatacola
10        Abbatemarco
11             Abbate
12             Abbott
13             Abbott
14        Abbruzzese
```

Working with Text Data

.str.split() method

- After we have extracted the First Name and Last Name, we can add a new column into the DataFrame using the insert() method.

```
In [197]: df.insert(0,column = "First Name" , value = firstName)
          df.insert(1,column = "Last Name" ,value = lastName)
          df.head()
```

Out[197]:

	First Name	Last Name	Name	Position Title	Department	Employee Annual Salary
0	Elvia	Aaron	Aaron, Elvia J	Water Rate Taker	Water Management	90744.0
1	Jeffery	Aaron	Aaron, Jeffery M	Police Officer	Police	84450.0
2	Karina	Aaron	Aaron, Karina	Police Officer	Police	84450.0
3	Kimberlei	Aaron	Aaron, Kimberlei R	Chief Contract Expediter	General Services	89880.0
4	Vicente	Abad Jr	Abad Jr, Vicente M	Civil Engineer Iv	Water Management	106836.0

- We can use value_counts() method to know which one is the most popular name.

```
In [199]: df["First Name"].value_counts().head()
```

```
Out[199]: Michael    1153
          John        899
          James       676
          Robert      622
          Joseph      537
          Name: First Name, dtype: int64
```

Working with Text Data

.str.split() method

- After we have extract First Name and Last Name, we can add a new column into the DataFrame using the insert() method.

```
In [197]: df.insert(0,column = "First Name" , value = firstName)
          df.insert(1,column = "Last Name" ,value = lastName)
          df.head()
```

Out[197]:

	First Name	Last Name	Name	Position Title	Department	Employee Annual Salary
0	Elvia	Aaron	Aaron, Elvia J	Water Rate Taker	Water Management	90744.0
1	Jeffery	Aaron	Aaron, Jeffery M	Police Officer	Police	84450.0
2	Karina	Aaron	Aaron, Karina	Police Officer	Police	84450.0
3	Kimberlei	Aaron	Aaron, Kimberlei R	Chief Contract Expediter	General Services	89880.0
4	Vicente	Abad Jr	Abad Jr, Vicente M	Civil Engineer Iv	Water Management	106836.0

Working with Text Data

.str.split() method

- In **split()**, **expand parameter** will expand the split strings into separate columns.

```
In [204]: df["Name"].str.split(",", expand=True).head()
```

Out[204]:

	0	1
0	Aaron	Elvia J
1	Aaron	Jeffery M
2	Aaron	Karina
3	Aaron	Kimberlei R
4	Abad Jr	Vicente M

- **split()** has **n parameter** which we can use to decide how many occurrences to split. The example shows that we are splitting strings around a space. Since, we give argument of 1 to the n parameter, it will only split the strings around the first space it finds.

```
In [208]: df["Position Title"].str.split(" ", expand=True, n = 1).head()
```

Out[208]:

	0	1
0	Water	Rate Taker
1	Police	Officer
2	Police	Officer
3	Chief	Contract Expediter
4	Civil	Engineer Iv

Working with Text Data

Regular Expression on text

- Regex is a powerful library to search for word(s) in the string. It can also find sequence of characters/words.
- **Example:** Validating a password format. When we want to create a new password, there are some requirements to be fulfilled like it should include uppercase, lowercase, numbers, and unique character.
- Python has a library specifically for regex.
- Python Regex Cheat Sheet : https://www.cheatography.com/mutanclan/cheat-sheets/python-regular-expression-regex/

Working with Text Data

Regular Expression on text

- In **re module**, there is a few common libraries used.
  - **match()** function returns a matched object if the text matches the pattern
  - **search()** function checks for a match anywhere in the string.
  - **compile()** Compiles a regular expression pattern into a regular expression object.
  - **findall()** Returns all non-overlapping matches of pattern in string, as a list of strings
- Tips: When you need to use an expression several times in a single program, using the compile() function to save the resulting regular expression object for reuse is more efficient.

Working with Text Data

Regular Expression on text

- dot (.) : Match any character except newline
- Search for names starting with “Ami”.

```
In [220]: text = "Amin Hakim"
          pattern = r"Ami."

          pattern = re.compile(pattern)

          result = pattern.search(text)
          print(result.group())
```

```
Amin
```

```
In [221]: text2 = "Amir Haiqal"

          pattern2 = pattern.search(text2)
          print(pattern2.group())
```

```
Amir
```

- bracket ([]) : Match anything in the bracket.
- For example, we search for digits in the text and return Boolean as the result.

```
In [223]: text3 = "I'm 21 years old"

          pattern = re.compile("[1-9]")
          result = pattern.search(text3)
          print(bool(result))
```

```
True
```

Working with Text Data

Regular Expression with Pandas

- In Pandas, we can use regex to search the pattern that we want in the text. As I mentioned before, we can include regex in **str.contains() function.**
- For example, we want to search for Position Title which has a digit in it.

```
In [213]: df.loc[df["Position Title"].str.contains("[1-9]")].head()
```

Out[213]:

	First Name	Last Name	Name	Position Title	Department	Employee Annual Salary
275	Mark	Akana	Akana, Mark S	Police Officer/Explsv Detect K9 Hndlr	Police	95178.0
892	Gilbert	Arce	Arce, Gilbert	Police Officer/Explsv Detect K9 Hndlr	Police	98016.0
1151	Rhonda	Avina	Avina, Rhonda S	Communications Operator I - 311	Oemc	59184.0
1233	Gayla	Baggett	Baggett, Gayla D	Communications Operator I - 311	Oemc	62004.0
1934	Timothy	Beran	Beran, Timothy	Police Officer/Explsv Detect K9 Hndlr	Police	91752.0

Working with Text Data

Regular Expression with Pandas

- We can go very specific with additional regex. Say, we want to search for Position Title, which contains the word, "Police" as well as number(s).
  - **dot (.)** : Match any character except newline
  - **star (*)** : Zero or more occurrences
  - **bracket ([])** : Match with anything in the bracket.

```
In [214]: df.loc[df["Position Title"].str.contains("Police.*[1-9]")]
```

Out[214]:

	First Name	Last Name	Name	Position Title	Department	Employee Annual Salary
275	Mark	Akana	Akana, Mark S	Police Officer/Explsv Detect K9 Hndlr	Police	95178.0
892	Gilbert	Arce	Arce, Gilbert	Police Officer/Explsv Detect K9 Hndlr	Police	98016.0
1934	Timothy	Beran	Beran, Timothy	Police Officer/Explsv Detect K9 Hndlr	Police	91752.0
2751	Michael	Bransfield	Bransfield, Michael P	Police Officer/Explsv Detect K9 Hndlr	Police	98016.0
3521	Sheila	Butler	Butler, Sheila	Police Officer/Explsv Detect K9 Hndlr	Police	95178.0
3561	James	Byrne	Byrne, James T	Police Officer/Explsv Detect K9 Hndlr	Police	98016.0
3764	Brian	Campbell	Campbell, Brian P	Police Officer/Explsv Detect K9 Hndlr	Police	95178.0

Working with Text Data

Regular Expression with Pandas

- Let's search for Position Title that starts with **Water** and ends with **Taker.**
  - **^** : Start of string
  - **Bracket []** : Match anything in the bracket. E.g: [Ww] means match any which have w or W
  - **Dot (.)** : Matches any single character except the newline character.
  - **star (*)** : Zero or more occurrences
  - **\b** : Word boundary. Spaces between word

```
In [215]: df.loc[df["Position Title"].str.contains("^[wW]ater.*Taker")]
```

Out[215]:

	First Name	Last Name	Name	Position Title	Department	Employee Annual Salary
0	Elvia	Aaron	Aaron, Elvia J	Water Rate Taker	Water Management	90744.0
2400	Brian	Bolton	Bolton, Brian E	Water Rate Taker	Water Management	78948.0
3143	Sharon	Brown	Brown, Sharon L	Water Rate Taker	Water Management	82728.0
3456	Carl	Burt	Burt, Carl S	Water Rate Taker	Water Management	90744.0
3626	Ophelia	Cage	Cage, Ophelia	Water Rate Taker	Water Management	90744.0
6954	Oscar	Diaz	Diaz, Oscar A	Water Rate Taker	Water Management	90744.0
7641	Patricia	Durant	Durant, Patricia B	Water Rate Taker	Water Management	90744.0

DATA ANALYSIS WITH PYTHON

Module 5: Group By

Group By Method

Introduction

- creates a category/chunk/segment from a DataFrame
- split data into unique groups from the variable/column of choice.
- For this module we will be using “fortune1000.csv” dataset.

In [109]:
```
import pandas as pd
fortune= pd.read_csv("data/fortune1000.csv", index_col="Rank")
fortune.head()
```

Out[109]:

Rank	Company	Sector	Industry	Location	Revenue	Profits	Employees
1	Walmart	Retailing	General Merchandisers	Bentonville, AR	482130	14694	2300000
2	Exxon Mobil	Energy	Petroleum Refining	Irving, TX	246204	16150	75600
3	Apple	Technology	Computers, Office Equipment	Cupertino, CA	233715	53394	110000
4	Berkshire Hathaway	Financials	Insurance: Property and Casualty (Stock)	Omaha, NE	210821	24083	331000
5	McKesson	Health Care	Wholesalers: Health Care	San Francisco, CA	181241	1476	70400

Group By Method

Introduction

- Using the **groupby()** method, Pandas will create a new object called *groupby.DataFrameGroupBy* object.
- **groupby object** only works if we apply a method on it.

```
In [110]: sectors = fortune.groupby("Sector")
          sectors

Out[110]: <pandas.core.groupby.groupby.DataFrameGroupBy object at 0x0000013AC407C940>
```

- We are creating an object based on Sector column. So, new information will be based on these unique groups from Sector columns.

```
In [26]: sectors.groups.keys()

Out[26]: dict_keys(['Aerospace & Defense', 'Apparel', 'Business Services', 'Chemicals', 'Energy', 'Engineering & Construction', 'Financi
         als', 'Food and Drug Stores', 'Food, Beverages & Tobacco', 'Health Care', 'Hotels, Resturants & Leisure', 'Household Products',
         'Industrials', 'Materials', 'Media', 'Motor Vehicles & Parts', 'Retailing', 'Technology', 'Telecommunications', 'Transportatio
         n', 'Wholesalers'])
```

Group By Method

Operations and Attributes with groupby Object

- **size()** : Number of rows in each group.

```
In [111]: sectors.size()
```

```
Out[111]: Sector
Aerospace & Defense            20
Apparel                        15
Business Services              51
Chemicals                      30
Energy                        122
Engineering & Construction     26
Financials                    139
Food and Drug Stores           15
Food, Beverages & Tobacco      43
Health Care                    75
Hotels, Resturants & Leisure   25
Household Products             28
Industrials                    46
```

- **first()** : Compute first of values within each group.
- The first value in the Aerospace & Defense Sector is Company **Boeing.** If we see in the list of Sector column, the first company is **Boeing** as well.

```
In [112]: sectors.first().head(2)
```

Out[112]:

Sector	Company	Industry	Location	Revenue	Profits	Employees
Aerospace & Defense	Boeing	Aerospace and Defense	Chicago, IL	96114	5176	161400
Apparel	Nike	Apparel	Beaverton, OR	30601	3273	62600

```
In [113]: fortune[fortune["Sector"] == "Aerospace & Defense"].head(2)
```

Out[113]:

Rank	Company	Sector	Industry	Location	Revenue	Profits	Employees
24	Boeing	Aerospace & Defense	Aerospace and Defense	Chicago, IL	96114	5176	161400
45	United Technologies	Aerospace & Defense	Aerospace and Defense	Farmington, CT	61047	7608	197200

Group By Method

Operations and Attributes with groupby Object

- **last()** : Compute last of group values
- The last Company in Aerospace & Defense Sector is **Delta Tucker Holdings**. If we go through the original DataFrame, we can see that the last company is also **Delta Tucker Holdings**

```
In [114]: sectors.last().head(2)
```

Out[114]:

Sector	Company	Industry	Location	Revenue	Profits	Employees
Aerospace & Defense	Delta Tucker Holdings	Aerospace and Defense	McLean, VA	1923	-133	12000
Apparel	Guess	Apparel	Los Angeles, CA	2204	82	13500

```
In [115]: fortune[fortune["Sector"] == "Aerospace & Defense"].tail(1)
```

Out[115]:

Rank	Company	Sector	Industry	Location	Revenue	Profits	Employees
987	Delta Tucker Holdings	Aerospace & Defense	Aerospace and Defense	McLean, VA	1923	-133	12000

Group By Method

Operations and Attributes with groupby Object

- **get_group() method**
  - Construct DataFrame of a specific group by specifying its name
  - For example, , I want to have a list of companies in the Retailing Sector.

- We can also use **masking technique.** The approach is different but the output is the same.

```
In [117]: sectors.get_group("Retailing")
```

Out[117]:

Rank	Company	Employees	Industry	Location	Profits	Revenue
1	Walmart	2300000	General Merchandisers	Bentonville, AR	14694	482130
15	Costco	161000	Specialty Retailers: Other	Issaquah, WA	2377	116199
28	Home Depot	385000	Specialty Retailers: Other	Atlanta, GA	7009	88519
38	Target	341000	General Merchandisers	Minneapolis, MN	3363	73785
47	Lowe's	225000	Specialty Retailers: Other	Mooresville, NC	2546	59074
71	Best Buy	125000	Specialty Retailers: Other	Richfield, MN	897	39745
89	TJX	216000	Specialty Retailers: Apparel	Framingham, MA	2278	30945
103	Macy's	157500	General Merchandisers	Cincinnati, OH	1072	27079
111	Sears Holdings	178000	General Merchandisers	Hoffman Estates, IL	-1129	25146
132	Staples	58963	Specialty Retailers: Other	Framingham, MA	379	21059

```
In [118]: fortune[fortune["Sector"] == "Retailing"]
```

Out[118]:

Rank	Company	Sector	Industry	Location	Revenue	Profits	Employees
1	Walmart	Retailing	General Merchandisers	Bentonville, AR	482130	14694	2300000
15	Costco	Retailing	Specialty Retailers: Other	Issaquah, WA	116199	2377	161000
28	Home Depot	Retailing	Specialty Retailers: Other	Atlanta, GA	88519	7009	385000
38	Target	Retailing	General Merchandisers	Minneapolis, MN	73785	3363	341000
47	Lowe's	Retailing	Specialty Retailers: Other	Mooresville, NC	59074	2546	225000
71	Best Buy	Retailing	Specialty Retailers: Other	Richfield, MN	39745	897	125000
89	TJX	Retailing	Specialty Retailers: Apparel	Framingham, MA	30945	2278	216000
103	Macy's	Retailing	General Merchandisers	Cincinnati, OH	27079	1072	157500
111	Sears Holdings	Retailing	General Merchandisers	Hoffman Estates, IL	25146	-1129	178000
132	Staples	Retailing	Specialty Retailers: Other	Framingham, MA	21059	379	58963

Group By Method

Operations and Attributes with groupby Object

- sum() method
  - returns a summation on each category for each other Numeric Columns

```
In [119]: sectors.sum()
```

Out[119]:

Sector	Revenue	Profits	Employees
Aerospace & Defense	357940	28742	968057
Apparel	95968	8236	346397
Business Services	272195	28227	1361050
Chemicals	243897	22628	463651
Energy	1517809	-73447	1188927
Engineering & Construction	153983	5304	406708
Financials	2217159	260209	3359948
Food and Drug Stores	483769	16759	1395398
Food, Beverages & Tobacco	555967	51417	1211632
Health Care	1614707	106114	2678289
Hotels, Resturants & Leisure	169546	20697	2484245
Household Products	234737	14428	646038

- mean() method
  - returns average value for each Numeric columns

```
In [120]: sectors.mean()
```

Out[120]:

Sector	Revenue	Profits	Employees
Aerospace & Defense	17897.000000	1437.100000	48402.850000
Apparel	6397.866667	549.066667	23093.133333
Business Services	5337.156863	553.470588	26687.254902
Chemicals	8129.900000	754.266667	15455.033333
Energy	12441.057377	-602.024590	9745.303279
Engineering & Construction	5922.423077	204.000000	15642.615385
Financials	15950.784173	1872.007194	24172.287770
Food and Drug Stores	32251.266667	1117.266667	93026.533333
Food, Beverages & Tobacco	12929.465116	1195.744186	28177.488372
Health Care	21529.426667	1414.853333	35710.520000
Hotels, Resturants & Leisure	6781.840000	827.880000	99369.800000
Household Products	8383.464286	515.285714	23072.785714

  - Extract the average of specific columns by specifying its name inside square brackets at the end.

```
In [121]: sectors.mean()["Revenue"]
```

```
Out[121]: Sector
Aerospace & Defense          17897.000000
Apparel                       6397.866667
Business Services             5337.156863
Chemicals                     8129.900000
Energy                       12441.057377
Engineering & Construction    5922.423077
Financials                   15950.784173
Food and Drug Stores         32251.266667
Food, Beverages & Tobacco    12929.465116
Health Care                  21529.426667
Hotels, Resturants & Leisure  6781.840000
Household Products            8383.464286
Industrials                  10816.978261
Materials                     6026.627907
Media                         8830.560000
```

Group By Method

Operations and Attributes with groupby Object

- count() method
  - Count data for every unique group. For example, according to the DataFrame below, we can know that there are 20 Employees who are working in Aerospace & Defense sector.

```
In [59]: sectors.count().head(10)
```

Out[59]:

Sector	Company	Industry	Location	Revenue	Profits	Employees
Aerospace & Defense	20	20	20	20	20	20
Apparel	15	15	15	15	15	15
Business Services	51	51	51	51	51	51
Chemicals	30	30	30	30	30	30
Energy	122	122	122	122	122	122
Engineering & Construction	26	26	26	26	26	26

Group By Method

Operations and Attributes with groupby Object

- **Grouping by multiple groups**
- In the previous example, we only group by one column/category. We can also do with multiple columns.
- Multiple groups give more detailed information. The return value is more spread out.
- In the example, we can see that there are 7 Industries in the Business Services sector. Meanwhile, there is only one industry in sector Aerospace & Defense.

```
In [124]: multiple = fortune.groupby(["Sector", "Industry"])
          multiple

Out[124]: <pandas.core.groupby.groupby.DataFrameGroupBy object at 0x0000013AC50F4470>
```

```
In [125]: multiple.sum()

Out[125]:
```

Sector	Industry	Revenue	Profits	Employees
Aerospace & Defense	Aerospace and Defense	357940	28742	968057
Apparel	Apparel	95968	8236	346397
Business Services	Advertising, marketing	22748	1549	124100
	Diversified Outsourcing Services	64829	4305	708330
	Education	7485	69	46755
	Financial Data Services	100778	17456	264926
	Miscellaneous	11185	2130	37720
	Temporary Help	34716	1000	60020
	Waste Management	30454	1718	119199
Chemicals	Chemicals	243897	22628	463651
Energy	Energy	67749	-13038	70072
	Mining, Crude-Oil Production	176435	-124555	149110
	Miscellaneous	3159	476	6700
	Oil and Gas Equipment, Services	82827	-4047	225795
	Petroleum Refining	705472	31888	244880
	Pipelines	130756	5814	73141

Group By Method

Operations and Attributes with groupby Object

- **agg() method a**ggregates using one or more operations over the specified axis.
- **dictionary** : it will do the specified operations on the columns.
- **list** : it will do all operations in the list.
- **combine both list and dictionary** : enable us to do specific operations on specific columns.

```
In [126]: sectors.agg({
              "Revenue" : "sum",
              "Profits" : "sum",
              "Employees" : "mean"
          })
```

Out[126]:

Sector	Revenue	Profits	Employees
Aerospace & Defense	357940	28742	48402.850000
Apparel	95968	8236	23093.133333
Business Services	272195	28227	26687.254902
Chemicals	243897	22628	15455.033333
Energy	1517809	-73447	9745.303279
Engineering & Construction	153983	5304	15642.615385
Financials	2217159	260209	24172.287770
Food and Drug Stores	483769	16759	93026.533333
Food, Beverages & Tobacco	555967	51417	28177.488372
Health Care	1614707	106114	35710.520000

Group By Method

Operations and Attributes with groupby Object

```
In [127]: sectors.agg(["sum", "mean","size" ])
```

Out[127]:

	Revenue			Profits			Employees		
	sum	mean	size	sum	mean	size	sum	mean	size
Sector									
Aerospace & Defense	357940	17897.000000	20	28742	1437.100000	20	968057	48402.850000	20
Apparel	95968	6397.866667	15	8236	549.066667	15	346397	23093.133333	15
Business Services	272195	5337.156863	51	28227	553.470588	51	1361050	26687.254902	51
Chemicals	243897	8129.900000	30	22628	754.266667	30	463651	15455.033333	30
Energy	1517809	12441.057377	122	-73447	-602.024590	122	1188927	9745.303279	122
Engineering & Construction	153983	5922.423077	26	5304	204.000000	26	406708	15642.615385	26
Financials	2217159	15950.784173	139	260209	1872.007194	139	3359948	24172.287770	139
Food and Drug Stores	483769	32251.266667	15	16759	1117.266667	15	1395398	93026.533333	15
Food, Beverages & Tobacco	555967	12929.465116	43	51417	1195.744186	43	1211632	28177.488372	43
Health Care	1614707	21529.426667	75	106114	1414.853333	75	2678289	35710.520000	75
Hotels, Resturants & Leisure	169546	6781.840000	25	20697	827.880000	25	2484245	99369.800000	25
Household Products	234737	8383.464286	28	14428	515.285714	28	646038	23072.785714	28
Industrials	497581	10816.978261	46	20764	451.391304	46	1545229	33591.934783	46

Group By Method

Iterating through Groups

- Create an empty DataFrame with the column names based on the original dataset.
- Using for loop, we can get all the information related to the highest revenue. (Its location, which company and industry it belongs to).

Create an empty DataFrame with column name based on original dataset

```
In [129]: df = pd.DataFrame(columns=fortune.columns)
          df
```

Out[129]:

Company	Sector	Industry	Location	Revenue	Profits	Employees

```
In [130]: for sector, data  in sectors:
              highest = data.nlargest(1, "Revenue")
              df = df.append(highest)
          df
```

Out[130]:

	Company	Sector	Industry	Location	Revenue	Profits	Employees
24	Boeing	Aerospace & Defense	Aerospace and Defense	Chicago, IL	96114	5176	161400
91	Nike	Apparel	Apparel	Beaverton, OR	30601	3273	62600
144	ManpowerGroup	Business Services	Temporary Help	Milwaukee, WI	19330	419	27000
56	Dow Chemical	Chemicals	Chemicals	Midland, MI	48778	7685	49495
2	Exxon Mobil	Energy	Petroleum Refining	Irving, TX	246204	16150	75600
155	Fluor	Engineering & Construction	Engineering, Construction	Irving, TX	18114	413	38758
4	Berkshire Hathaway	Financials	Insurance: Property and Casualty (Stock)	Omaha, NE	210821	24083	331000
7	CVS Health	Food and Drug Stores	Food and Drug Stores	Woonsocket, RI	153290	5237	199000
41	Archer Daniels Midland	Food, Beverages & Tobacco	Food Production	Chicago, IL	67702	1849	32300
5	McKesson	Health Care	Wholesalers: Health Care	San Francisco, CA	181241	1476	70400
109	McDonald's	Hotels, Resturants & Leisure	Food Services	Oak Brook, IL	25413	4529	420000

DATA ANALYSIS WITH PYTHON

Module 6: MultiIndex

MultiIndex

Introduction

- Enables you to store and manipulate data with an arbitrary number of dimensions in lower-dimensional data structures like Series (1d) and DataFrame (2d).
- Categorize the data more effectively.
- Examples :

		Price in US Dollars
Date	Country	
	Argentina	2.39
	Australia	3.74
	Brazil	3.35
	Britain	4.22
	Canada	4.14
	Chile	2.94
	China	2.68
	Colombia	2.43
	Costa Rica	4.02

MultiIndex

Summarisation on this dataset

- **No null values** for every column.
- The Date column is converted into datetime type by using **parameter parse_dates**. We can do operations such as add date, subtract date and many more on this column
- All column types are precisely according to their data types.

```
In [1]: import pandas as pd
        bigmac = pd.read_csv("data/bigmac.csv", parse_dates=["Date"])
        bigmac.head()
```

Out[1]:

	Date	Country	Price in US Dollars
0	2016-01-01	Argentina	2.39
1	2016-01-01	Australia	3.74
2	2016-01-01	Brazil	3.35
3	2016-01-01	Britain	4.22
4	2016-01-01	Canada	4.14

MultiIndex

1. Create a MultiIndex DataFrame with **.set_index()** method

- Then, we can sort both indexes.

```
In [5]: bigmac.sort_index(inplace=True)

In [6]: bigmac.head()
```

Out[6]:

		Price in US Dollars
Date	Country	
2010-01-01	Argentina	1.84
	Australia	3.98
	Brazil	4.76
	Britain	3.67
	Canada	3.97

MultiIndex

1. Create a MultiIndex DataFrame with **.set_index()** method

- Another way we can set the MultiIndex is by using the index_col parameter in **read_csv() method**.
- The output is the same, but we can complete this in one single line.
- As you can see, the column is automatically categorized into Date and Country. Since Date is first in the list, then the first column will be Date.

```
In [40]: bigmac = pd.read_csv("data/bigmac.csv", index_col=["Date", "Country"], parse_dates=["Date"])
         bigmac.sort_index(inplace=True)
         bigmac.head(10)
```

Out[40]:

		Price in US Dollars
Date	**Country**	
2010-01-01	**Argentina**	1.84
	Australia	3.98
	Brazil	4.76
	Britain	3.67
	Canada	3.97
	Chile	3.18
	China	1.83
	Colombia	3.91
	Costa Rica	3.52
	Czech Republic	3.71

MultiIndex

1. Create a MultiIndex DataFrame with **.set_index()** method

- we can use type() built-in function to check our DataFrame index.

```
In [9]: type(bigmac.index)
Out[9]: pandas.core.indexes.multi.MultiIndex

In [10]: bigmac.index.names
Out[10]: FrozenList(['Date', 'Country'])
```

- To extract information on every row, we have to give two data of Date and Country.

```
In [11]: bigmac.index[0]
Out[11]: (Timestamp('2010-01-01 00:00:00'), 'Argentina')

In [12]: bigmac.loc["2016-01-01", "Italy"]
Out[12]: Price in US Dollars    4.3
         Name: (2016-01-01 00:00:00, Italy), dtype: float64
```

MultiIndex

2. **get_levels_values()** method

- returns an Index of values for requested level.
- **Parameters**
  - level : level is either the integer position of the level in the MultiIndex or the name of the level.
- For this dataset, 0 refers to the **Date column,** which is the first level of MultiIndex. Meanwhile, 1 refers to the 2nd level which is **Country column**.

```
In [13]: bigmac.index.get_level_values(level = 0)

Out[13]: DatetimeIndex(['2010-01-01', '2010-01-01', '2010-01-01', '2010-01-01',
                        '2010-01-01', '2010-01-01', '2010-01-01', '2010-01-01',
                        '2010-01-01', '2010-01-01',
                        ...
                        '2016-01-01', '2016-01-01', '2016-01-01', '2016-01-01',
                        '2016-01-01', '2016-01-01', '2016-01-01', '2016-01-01',
                        '2016-01-01', '2016-01-01'],
                       dtype='datetime64[ns]', name='Date', length=652, freq=None)

In [14]: bigmac.index.get_level_values(level = 1)

Out[14]: Index(['Argentina', 'Australia', 'Brazil', 'Britain', 'Canada', 'Chile',
                'China', 'Colombia', 'Costa Rica', 'Czech Republic',
                ...
                'Switzerland', 'Taiwan', 'Thailand', 'Turkey', 'UAE', 'Ukraine',
                'United States', 'Uruguay', 'Venezuela', 'Vietnam'],
               dtype='object', name='Country', length=652)
```

MultiIndex

3. **set_names()** method on MultiIndex

- Set Index or MultiIndex name.
- Able to set new names partially and by level.
- *Hint* : Another approach you can use is to rename the columns first before setting them as index.

```
In [15]: bigmac.index.set_names(["Day", "Location"], inplace=True)
         bigmac.head()
```

Out[15]:

		Price in US Dollars
Day	**Location**	
2010-01-01	**Argentina**	1.84
	Australia	3.98
	Brazil	4.76
	Britain	3.67
	Canada	3.97

MultiIndex

4. sort_index() Method

- sort_index() method enables us to sort object accordingly.
- For example, we can sort the Date in ascending and Country in Descending order. This can be done by passing a list of Boolean to the parameter.
- There are other parameters that we can use.

```
In [17]: bigmac.sort_index(ascending=[True, False])
```

Out[17]:

		Price in US Dollars
Date	Country	
2010-01-01	Uruguay	3.32
	United States	3.58
	Ukraine	1.83
	UAE	2.99
	Turkey	3.83
	Thailand	2.11
	Taiwan	2.36
	Switzerland	6.30
	Sweden	5.51
	Sri Lanka	1.83
	South Korea	2.98
	South Africa	2.46

MultiIndex

5. Extract rows from MultiIndex DataFrame

- In the previous module, loc and iloc are used to extract data by rows. For MultiIndex, loc and iloc can also be used but we need to pass additional data.
- Since the DataFrame is using two indexes, we need to use two arguments to extract specific information. Those two arguments need to be in tuple format.

```
In [18]: bigmac.head(3)
```

Out[18]:

		Price in US Dollars
Date	Country	
2010-01-01	Argentina	1.84
	Australia	3.98
	Brazil	4.76

```
In [19]: bigmac.loc[("2010-01-01","Argentina"),"Price in US Dollars"]
```

```
Out[19]: Date        Country
         2010-01-01  Argentina    1.84
         Name: Price in US Dollars, dtype: float64
```

MultiIndex

5. Extract rows from MultiIndex DataFrame

- Here is an example of how we can extract data from Price in US Dollar column at a specific date. Since Country is also an index, by default it will appear as well.

```
In [21]: bigmac.loc[("2010-01-01"),"Price in US Dollars"]
```

Out[21]:

Date	Country	
2010-01-01	Argentina	1.84
	Australia	3.98
	Brazil	4.76
	Britain	3.67
	Canada	3.97
	Chile	3.18
	China	1.83
	Colombia	3.91
	Costa Rica	3.52
	Czech Republic	3.71
	Denmark	5.99
	Egypt	2.38
	Euro area	4.84
	Hong Kong	1.91
	Hungary	3.86
	Indonesia	2.24
	Israel	3.99
	Japan	3.50

MultiIndex

6. transpose() method

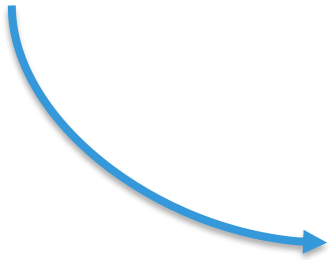
- Pandas DataFrame.transpose() function transposes index and columns of the dataFrame.
- It reflects the DataFrame over its main diagonal by writing rows as columns and vice-versa.
- Bigmac DataFrame has 652 rows and one column. After applying the transpose() method, the row and column size changed.

```
In [22]: bigmac
```

Out[22]:

		Price in US Dollars
Date	Country	
2010-01-01	Argentina	1.84
	Australia	3.98
	Brazil	4.76
	Britain	3.67
	...	
	United States	4.93
	Uruguay	3.74
	Venezuela	0.66
	Vietnam	2.67

652 rows × 1 columns



```
In [23]: bigmac.transpose()
```

Out[23]:

Date	2010-01-01			
Country	Argentina	Australia	Brazil	Britain
Price in US Dollars	1.84	3.98	4.76	3.67

1 rows × 652 columns

MultiIndex

7. **swaplevel()** method

- Swap index from one level to another.
- If we have 3 to 4 levels MultiIndex, then, i and j refer to the index level.
- **Swap levels** i and j in a **MultiIndex** on a particular axis.
- After applying **swaplevel() method**, the first index and second index swap with each other.

```
In [26]: bigmac.head()
```

Out[26]:

		Price in US Dollars
Date	Country	
2010-01-01	Argentina	1.84
	Australia	3.98
	Brazil	4.76
	Britain	3.67
	Canada	3.97

```
In [27]: bigmac.swaplevel(i = "Date", j ="Country")
```

Out[27]:

		Price in US Dollars
Country	Date	
Argentina	2010-01-01	1.84
Australia	2010-01-01	3.98
Brazil	2010-01-01	4.76

MultiIndex

8. **stack()** method

- Returns a reshaped DataFrame or Series having a multi-level index with one or more new inner-most levels compared to the current DataFrame.
- The new inner-most levels are created by pivoting the columns of the current dataFrame:
  - change Pandas DataFrame into Pandas Series
  - All the columns become the row

```
In [26]: bigmac.head()
```

Out[26]:

		Price in US Dollars
Date	Country	
2010-01-01	Argentina	1.84
	Australia	3.98
	Brazil	4.76
	Britain	3.67
	Canada	3.97

```
In [28]: bigmac.stack()
```

```
Out[28]: Date        Country
         2010-01-01  Argentina       Price in US Dollars    1.84
                     Australia       Price in US Dollars    3.98
                     Brazil          Price in US Dollars    4.76
                     Britain         Price in US Dollars    3.67
                     Canada          Price in US Dollars    3.97
                     Chile           Price in US Dollars    3.18
                     China           Price in US Dollars    1.83
                     Colombia        Price in US Dollars    3.91
                     Costa Rica      Price in US Dollars    3.52
                     Czech Republic  Price in US Dollars    3.71
                     Denmark         Price in US Dollars    5.99
                     Egypt           Price in US Dollars    2.38
                     Euro area       Price in US Dollars    4.84
```

MultiIndex

8. **stack()** method

- Using method chaining, we can apply **to_frame()** to turn the Series into DataFrame.

```
In [29]: bigmac.stack().to_frame()
```

Out[29]:

			0
Date	**Country**		
2010-01-01	**Argentina**	**Price in US Dollars**	1.84
	Australia	**Price in US Dollars**	3.98
	Brazil	**Price in US Dollars**	4.76
	Britain	**Price in US Dollars**	3.67
	Canada	**Price in US Dollars**	3.97
	Chile	**Price in US Dollars**	3.18
	China	**Price in US Dollars**	1.83

MultiIndex

8. **stack()** method

- Here's another example using a different dataset named **worldstats.** In this dataset, we assign country and year as Index.
- We are left with two columns which are Population and GDP. Applying stack() method will change the remaining columns to rows.

```
In [30]: world = pd.read_csv("data/worldstats.csv", index_col=["country", "year"])
         world.head()
```

Out[30]:

country	year	Population	GDP
Arab World	2015	392022276.0	2.530102e+12
	2014	384222592.0	2.873600e+12
	2013	376504253.0	2.846994e+12
	2012	368802611.0	2.773270e+12
	2011	361031820.0	2.497945e+12

```
In [31]: world = world.stack().to_frame()
         world
```

Out[31]:

			0
country	year		
Arab World	2015	Population	3.920223e+08
		GDP	2.530102e+12
	2014	Population	3.842226e+08
		GDP	2.873600e+12
	2013	Population	3.765043e+08
		GDP	2.846994e+12

MultiIndex

9. unstack() method

- Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.
- It is the opposite of stack() method.
- It changes rows into columns.

```
In [33]: world.head()
```

Out[33]:

			0
country	year		
Arab World	2015	Population	3.920223e+08
		GDP	2.530102e+12
	2014	Population	3.842226e+08
		GDP	2.873600e+12
	2013	Population	3.765043e+08

```
In [34]: world.unstack()
```

Out[34]:

		0	
		Population	GDP
country	year		
Afghanistan	1960	8994793.0	5.377778e+08
	1961	9164945.0	5.488889e+08
	1962	9343772.0	5.466667e+08
	1963	9531555.0	7.511112e+08
	1964	9728645.0	8.000000e+08

MultiIndex

9. **unstack()** method

- Two times unstack() method.

```
In [35]: world.unstack().unstack()
```

Out[35]:

	0									
	Population									
year	1960	1961	1962	1963	1964	1965	1966	1967	1968	1969
country										
Afghanistan	8.994793e+06	9.164945e+06	9.343772e+06	9.531555e+06	9.728645e+06	9.935358e+06	1.014884e+07	1.036860e+07	1.059979e+07	1.084951e+07
Albania	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Algeria	1.112489e+07	1.140486e+07	1.169015e+07	1.198513e+07	1.229597e+07	1.262695e+07	1.298027e+07	1.335420e+07	1.374438e+07	1.414444e+07
Andorra	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Angola	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Antigua and Barbuda	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

MultiIndex

9. **unstack()** method

- Three times unstack() method.
- Since the index is not a MultiIndex, the output will be a Series.
- The levels involved will automatically get sorted.

```
In [36]: world.unstack().unstack().unstack()
```

Out[36]:

		year	country	
0	Population	1960	Afghanistan	8.994793e+06
			Albania	NaN
			Algeria	1.112489e+07
			Andorra	NaN
			Angola	NaN
			Antigua and Barbuda	NaN
			Arab World	NaN
			Argentina	NaN
			Armenia	NaN
			Aruba	NaN
			Australia	1.027648e+07
			Austria	7.047539e+06
			Azerbaijan	NaN
			Bahamas, The	1.095260e+05
			Bahrain	NaN
			Bangladesh	4.820070e+07
			Barbados	NaN
			Belarus	NaN
			Belgium	9.153489e+06
			Belize	9.206800e+04
			Benin	2.431620e+06
			Bermuda	4.440000e+04
			Bhutan	NaN
			Bolivia	3.693451e+06
			Bosnia and Herzegovina	NaN
			Botswana	5.240290e+05
			Brazil	7.249358e+07
			Brunei Darussalam	NaN
			Bulgaria	NaN
			Burkina Faso	4.829291e+06

MultiIndex

10. **pivot()** method

- Reshape data (produce a “pivot” table) based on column values.
- Before we apply the method, let's understand the dataset first.
  - There are only five unique salesmen.
  - There are 1830 rows in total.

```
In [42]: sales = pd.read_csv("data/salesmen.csv", parse_dates=["Date"])
         sales.head(3)
```

Out[42]:

	Date	Salesman	Revenue
0	2016-01-01	Bob	7172
1	2016-01-02	Bob	6362
2	2016-01-03	Bob	5982

```
In [43]: len(sales)
```

```
Out[43]: 1830
```

```
In [44]: sales["Salesman"].value_counts()
```

```
Out[44]: Bob       366
         Dave      366
         Jeb       366
         Oscar     366
         Ronald    366
         Name: Salesman, dtype: int64
```

MultiIndex

10. **pivot()** method

- There is a better way to visualize the dataset in table. A good table makes understanding the data much easier. From the table below, we can identify who has the highest revenue for each day. This method allows us to understand the table from different angles/perspectives.
  - Index : Column to make as the new frame's index.
  - Columns : Column to make as the new frame's columns.
  - Values : Column(s) to populate new frame's values.

```
In [45]: sales.pivot(index="Date", columns="Salesman", values="Revenue")
```

Out[45]:

Salesman	Bob	Dave	Jeb	Oscar	Ronald
Date					
2016-01-01	7172	1864	4430	5250	2639
2016-01-02	6362	8278	8026	8661	4951
2016-01-03	5982	4226	5188	7075	2703
2016-01-04	7917	3868	3144	2524	4258
2016-01-05	7837	2287	938	2793	7771
2016-01-06	1744	7859	8702	7794	5930
2016-01-07	918	8597	4250	9728	1933
2016-01-08	9863	3092	9719	5263	5709
2016-01-09	8337	1794	5614	7144	4707
2016-01-10	7543	7105	301	7663	8267
2016-01-11	1053	6851	9489	8888	1340
2016-01-12	4362	7147	8719	3092	279
2016-01-13	6812	6160	2349	6139	7540
2016-01-14	9047	7533	7690	5235	4214
2016-01-15	3594	9883	6917	5507	5346
2016-01-16	9800	3622	474	4947	9636

MultiIndex

11. pivot_table() method

- Create a spreadsheet-styled pivot table as a DataFrame.
- **parameters**
  - values : column to aggregate the columns that we used to calculate the values from.
  - index : new DataFrame will be based on the category of this column
  - aggfunc : by default, it is mean. Can change into sum, max, min, etc

```
In [46]: sales.head(3)
```

Out[46]:

	Date	Salesman	Revenue
0	2016-01-01	Bob	7172
1	2016-01-02	Bob	6362
2	2016-01-03	Bob	5982

```
In [47]: sales.pivot_table(values="Revenue", index="Salesman", aggfunc="mean")
```

Out[47]:

	Revenue
Salesman	
Bob	4992.292350
Dave	5079.407104
Jeb	5241.579235
Oscar	4857.319672
Ronald	4992.109290

MultiIndex

11. pivot_table() method

- Using the groupby() method, we can also get the same result.

```
In [48]: sales.groupby("Salesman")["Revenue"].mean().to_frame()
```

Out[48]:

Salesman	Revenue
Bob	4992.292350
Dave	5079.407104
Jeb	5241.579235
Oscar	4857.319672
Ronald	4992.109290

MultiIndex

11. pivot_table() method

- Let's try pivoting the new table based on Revenue column
  - **index = ["Gender" , "Item"]** to categorise the spending into two genders as well as items
  - **aggfunc = "mean"** means we want to know the mean value of the value column. (mean spending of an item between the gender)
  - **values ="Spend"** is the column which aggregate function will apply to.

```
In [49]: foods = pd.read_csv("data/foods.csv")
         foods.head(3)
```

Out[49]:

	First Name	Gender	City	Frequency	Item	Spend
0	Wanda	Female	Stamford	Weekly	Burger	15.66
1	Eric	Male	Stamford	Daily	Chalupa	10.56
2	Charles	Male	New York	Never	Sushi	42.14

```
In [50]: foods.pivot_table(index=["Gender", "Item"], values="Spend", aggfunc="mean")
```

Out[50]:

		Spend
Gender	Item	
Female	Burger	49.930488
	Burrito	50.092000
	Chalupa	54.635000
	Donut	49.926316
	Ice Cream	49.788519
	Sushi	50.355699
Male	Burger	49.613919
	Burrito	48.344819
	Chalupa	49.186761
	Donut	43.649565
	Ice Cream	51.096000
	Sushi	55.614384

MultiIndex

12. melt() method

- Opposite of pivot() method. This method **condenses the columns** and creates more rows..

```
In [52]: df = foods.pivot_table(index=["Item"], values="Spend", aggfunc="max", columns="City")
         df
```

Out[52]:

City	New York	Philadelphia	Stamford
Item			
Burger	98.96	99.68	97.20
Burrito	98.04	96.79	99.21
Chalupa	98.43	99.29	99.87
Donut	95.63	96.52	99.26
Ice Cream	97.83	99.24	99.17
Sushi	99.51	99.02	98.48

- First, we have to reset the index. The index for DataFrame above is labelled with Item.

```
In [53]: df.reset_index(inplace=True)
         df
```

Out[53]:

City	Item	New York	Philadelphia	Stamford
0	Burger	98.96	99.68	97.20
1	Burrito	98.04	96.79	99.21
2	Chalupa	98.43	99.29	99.87
3	Donut	95.63	96.52	99.26
4	Ice Cream	97.83	99.24	99.17
5	Sushi	99.51	99.02	98.48

MultiIndex

12. melt() method

- After applying the .melt() method, we can see that the three columns of City became one.
- Parameter:
  - id_vars : the variables that we don't want to change.

```
In [53]: df
```

Out[53]:

City	Item	New York	Philadelphia	Stamford
0	Burger	98.96	99.68	97.20
1	Burrito	98.04	96.79	99.21
2	Chalupa	98.43	99.29	99.87
3	Donut	95.63	96.52	99.26
4	Ice Cream	97.83	99.24	99.17
5	Sushi	99.51	99.02	98.48

```
In [54]: df.melt(id_vars="Item")
```

Out[54]:

Item	City	value
0	Burger	New York	98.96
1	Burrito	New York	98.04
2	Chalupa	New York	98.43
3	Donut	New York	95.63
4	Ice Cream	New York	97.83
5	Sushi	New York	99.51
6	Burger	Philadelphia	99.68
7	Burrito	Philadelphia	96.79
8	Chalupa	Philadelphia	99.29
9	Donut	Philadelphia	96.52
10	Ice Cream	Philadelphia	99.24
11	Sushi	Philadelphia	99.02
12	Burger	Stamford	97.20
13	Burrito	Stamford	99.21

DATA ANALYSIS WITH PYTHON

Module 7: Merging, Joining, and Concatenating

Merging, Joining, and Concatenating

Introduction

- How to join, merge and concatenate multiple DataFrame.
- Works like SQL. Combine different sources on unique values.
- In this module, we will be working with five different CSV files which can be linked together.

Merging, Joining, and Concatenating

Datasets

- Week 1 Dataset

```
week1 = pd.read_csv("data/Restaurant - Week 1 Sales.csv")
week1.head(3)
```

	Customer ID	Food ID
0	537	9
1	97	4
2	658	1

- Week 2 Dataset

```
week2 = pd.read_csv("data/Restaurant - Week 2 Sales.csv")
week2.head(3)
```

	Customer ID	Food ID
0	688	10
1	813	7
2	495	10

  - **Customer ID** : unique number given to each customer.
  - **Food ID** : unique number for each food. This dataset shows food which the customers bought in Week 1.

- The same structure as the above dataFrame. This is data for 2nd week.

Merging, Joining, and Concatenating

Datasets

- Foods Dataset

```
In [32]: foods = pd.read_csv("data/Restaurant - Foods.csv")
         foods.head(3)
```

Out[32]:

	Food ID	Food Item	Price
0	1	Sushi	3.99
1	2	Burrito	9.99
2	3	Taco	2.99

  - This dataset store information on each food. It includes Food names and prices.

- Customer Dataset

```
In [33]: customers = pd.read_csv("data/Restaurant - Customers.csv")
         customers.head(3)
```

Out[33]:

	ID	First Name	Last Name	Gender	Company	Occupation
0	1	Joseph	Perkins	Male	Dynazzy	Community Outreach Specialist
1	2	Jennifer	Alvarez	Female	DabZ	Senior Quality Engineer
2	3	Roger	Black	Male	Tagfeed	Account Executive

  - This dataset stores information about customers. . The ID in this dataset will be the key to link with other datasets.

Merging, Joining, and Concatenating

1. **pd.concat()** method

- Since week1 comes before week2, then week2 dataset will be appended to week1.
- After applying the concat method, the last index number is 249, even though we have 500 rows.
- This is because Pandas use the original index number. We can solve this by setting the **ignore_index parameter** to true.

```
In [34]: len(pd.concat([week1,week2]))
Out[34]: 500

In [35]: pd.concat([week1,week2])
```

Out[35]:

	Customer ID	Food ID
0	537	9
1	97	4
2	658	1
3	202	2
4	155	9
5	213	8
6	600	1
7	503	5
8	71	3
9	174	3
...	...	...
248	252	9
249	249	6

500 rows × 2 columns

Merging, Joining, and Concatenating

1. **pd.concat()** method

- Using **ignore_index** parameter, we combine the datasets and the index counts will follow the rows location number.

```
In [36]: df = pd.concat([week1,week2], ignore_index=True)
         df
```

Out[36]:

	Customer ID	Food ID
0	537	9
1	97	4
2	658	1
3	202	2
4	155	9
5	213	8
6	600	1
...	...	...
496	556	10
497	547	9
498	252	9
499	249	6

500 rows × 2 columns

Merging, Joining, and Concatenating

1. **pd.concat()** method

- **keys parameter** enables us to label the dataset that we concat /combine.
- It creates a MultiIndex DataFrame in which the first index is the keys and the second index is the original index.

```
In [43]: sales = pd.concat([week1,week2], keys=["Week 1","Week 2"])
         sales
```

Out[43]:

		Customer ID	Food ID
Week 1	0	537	9
	1	97	4
	2	658	1
	3	202	2
	4	155	9
	5	213	8
	6	600	1
	7	503	5
	8	71	3
	9	174	3
	10	961	9
	11	966	5
	12	641	4
	13	288	2
	14	149	4
	15	954	2
	16	147	9
	17	155	1

Merging, Joining, and Concatenating

2. **append()** method

- Append rows of two DataFrame.
- It produces the same output as the concat() method.
- The **.append()** method also has **ignore_index parameter.**

```
In [46]: week1.append(week2, ignore_index=True)
```

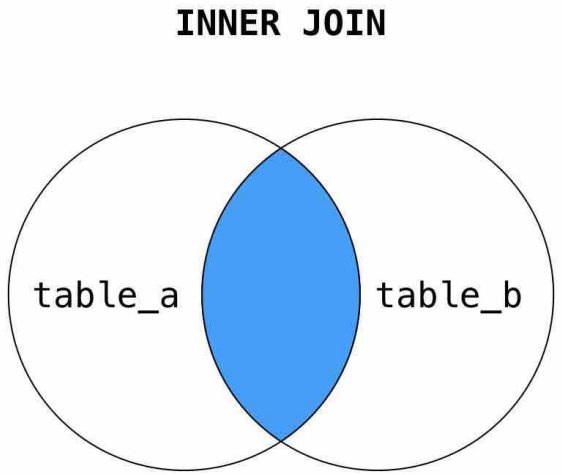
Out[46]:

	Customer ID	Food ID
0	537	9
1	97	4
2	658	1
3	202	2
4	155	9
5	213	8
6	600	1
7	503	5
8	71	3
9	174	3
10	961	9
11	966	5
12	641	4
13	288	2
14	149	4
15	954	2
16	147	9
17	155	1

Merging, Joining, and Concatenating

3. Inner join

- From the image, the inner join is the intersection of two datasets. The intersection is two datasets that share the universal reference point(s) or key.
- Inner join is used widely in SQL language or database language. It can also be used in Pandas library. This method is handy to combine two datasets with similar reference points/keys.




```
SELECT column_name
FROM table_a
INNER JOIN table_b
ON table_a.col_name = table_b.col_name;
```

dyclassroom.com

Merging, Joining, and Concatenating

3. Inner join

- **left** parameter : the first table
- **right** parameter : the second table
- **on** : the reference point or key
- **how** parameter : the method. Here, we specify inner. (We will learn about outer, left and right soon)
- **suffixes** parameter : Suffix to apply to overlapping column names in the left and right side, respectively.

```
df= pd.merge(left=week1, right=week2, how="inner", on="Customer ID", suffixes=[" - Week 1", " - Week 2"])
df
```

	Customer ID	Food ID - Week 1	Food ID - Week 2
0	537	9	5
1	155	9	3
2	155	1	3
3	503	5	8
4	503	5	9
5	550	6	7
6	101	7	4
7	75	6	4
8	77	1	7
9	77	9	7
10	77	2	7
11	77	9	7
12	628	4	7
13	798	9	5
14	798	1	5
15	462	3	8
16	304	3	3

Merging, Joining, and Concatenating

3. Inner join

- If you notice, there are **two rows** that have the **Customer ID** of **155**.
- This is because there are two rows of Customer ID 15 in Week 1 dataset. This means **Customer 155 came twice in week 1.** Hence, when we do inner join, it will duplicate the value.

```
In [66]: df.head()
```

Out[66]:

	Customer ID	Food ID - Week 1	Food ID - Week 2
0	537	9	5
1	155	9	3
2	155	1	3
3	503	5	8
4	503	5	9

```
In [68]: week1[week1["Customer ID"] == 155]
```

Out[68]:

	Customer ID	Food ID
4	155	9
17	155	1

```
In [69]: week2[week2["Customer ID"] == 155]
```

Out[69]:

	Customer ID	Food ID
208	155	3

Merging, Joining, and Concatenating

3. Inner join

- Extract customers who came on both weeks and bought the same food.
- However, we can still see some duplicates in the DataFrame (Customer 21 and 578). Why?

```
In [71]: pd.merge(left=week1, right=week2, how="inner" , on=["Customer ID" , "Food ID"])
```

Out[71]:

	Customer ID	Food ID
0	304	3
1	540	3
2	937	10
3	233	3
4	21	4
5	21	4
6	922	1
7	578	5
8	578	5

- *Answer: These customers came twice either in Week 1 or Week 2. Try extract the data from weeks 1 and 2 and compare them. You will understand more!*

Merging, Joining, and Concatenating

4. Outer Joins

```
In [73]: merged = pd.merge(left=week1, right=week2, how="outer",on="Customer ID", suffixes=[" - Week 1", " - Week 2"], indicator=True)
         merged
```

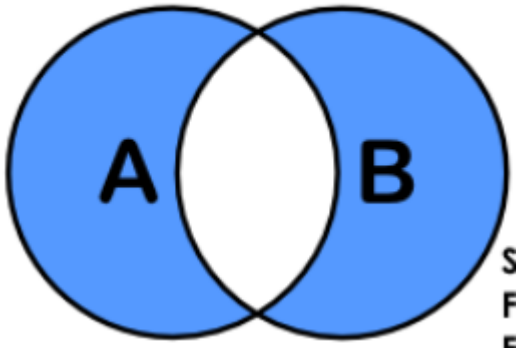
Out[73]:

	Customer ID	Food ID - Week 1	Food ID - Week 2	_merge
0	537	9.0	5.0	both
1	97	4.0	NaN	left_only
2	658	1.0	NaN	left_only
3	202	2.0	NaN	left_only
4	155	9.0	3.0	both
5	155	1.0	3.0	both
6	213	8.0	NaN	left_only
7	600	1.0	NaN	left_only
8	503	5.0	8.0	both
9	503	5.0	9.0	both
10	71	3.0	NaN	left_only
11	71	8.0	NaN	left_only
12	174	3.0	NaN	left_only
13	961	9.0	NaN	left_only
14	966	5.0	NaN	left_only

Merging, Joining, and Concatenating

4. Outer Joins (with exception)

- First, we have to combine all the datasets using one intersection point.
- Then, we remove the points which intersect between these two datasets



Merging, Joining, and Concatenating

4. Outer Joins (with exception)

```
In [76]: mask = merged["_merge"].isin(["right_only" , "left_only"])
         merged[mask]
```

Out[76]:

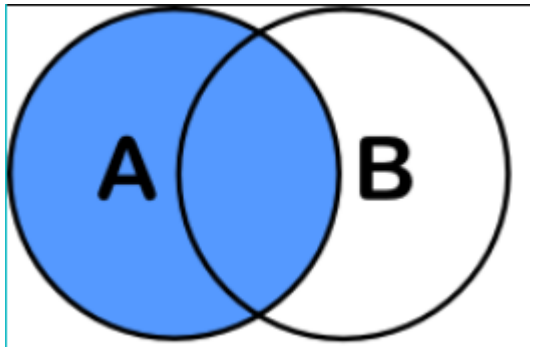
	Customer ID	Food ID - Week 1	Food ID - Week 2	_merge
1	97	4.0	NaN	left_only
2	658	1.0	NaN	left_only
3	202	2.0	NaN	left_only
6	213	8.0	NaN	left_only
7	600	1.0	NaN	left_only
10	71	3.0	NaN	left_only
11	71	8.0	NaN	left_only
12	174	3.0	NaN	left_only
13	961	9.0	NaN	left_only
14	966	5.0	NaN	left_only
15	641	4.0	NaN	left_only
16	288	2.0	NaN	left_only
17	149	4.0	NaN	left_only
439	940	NaN	8.0	right_only
440	571	NaN	7.0	right_only
441	888	NaN	2.0	right_only
442	664	NaN	6.0	right_only
443	143	NaN	4.0	right_only
444	505	NaN	3.0	right_only
445	54	NaN	8.0	right_only
446	367	NaN	8.0	right_only
447	883	NaN	8.0	right_only
448	251	NaN	9.0	right_only
449	855	NaN	4.0	right_only
450	559	NaN	10.0	right_only
451	276	NaN	4.0	right_only
452	556	NaN	10.0	right_only
453	252	NaN	9.0	right_only

392 rows × 4 columns

Merging, Joining, and Concatenating

5. Left Joins

- returns all rows from the **left** table, AND the intersection between these two tables
- Focus on one dataFrame and include intersection on the second database.



Merging, Joining, and Concatenating

5. Left Joins

```
In [80]: week1.head(3)
```

Out[80]:

	Customer ID	Food ID
0	537	9
1	97	4
2	658	1

```
In [81]: foods.head(3)
```

Out[81]:

	Food ID	Food Item	Price
0	1	Sushi	3.99
1	2	Burrito	9.99
2	3	Taco	2.99

```
In [83]: df = pd.merge(left=week1, right=foods, how="left", on="Food ID")
         df
```

Out[83]:

	Customer ID	Food ID	Food Item	Price
0	537	9	Donut	0.99
1	97	4	Quesadilla	4.25
2	658	1	Sushi	3.99
3	202	2	Burrito	9.99
4	155	9	Donut	0.99
5	213	8	Salad	11.25
6	600	1	Sushi	3.99
7	503	5	Pizza	2.49
8	71	3	Taco	2.99
9	174	3	Taco	2.99
10	961	9	Donut	0.99
11	966	5	Pizza	2.49
12	641	4	Quesadilla	4.25
13	288	2	Burrito	9.99
14	149	4	Quesadilla	4.25
15	954	2	Burrito	9.99
16	147	9	Donut	0.99
17	155	1	Sushi	3.99
18	550	6	Pasta	13.99
19	101	7	Steak	24.99

- Let's combine both of the tables.
- Now, we can identify the name of the food the customers eat.

Merging, Joining, and Concatenating

5. Left Joins

- In the **merge() method,** we have an additional parameter which is called **sort**.
- By default, it set to False. If it is True, it will sort the keys in ascending order.

```
In [82]: pd.merge(left=week1, right=foods, on="Food ID", sort=True)
```

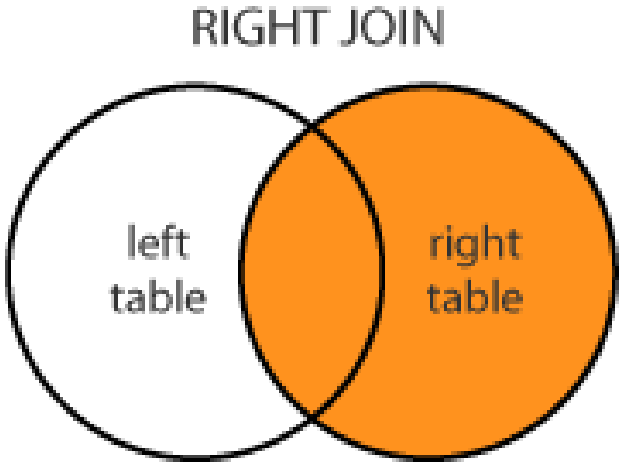
Out[82]:

	Customer ID	Food ID	Food Item	Price
0	658	1	Sushi	3.99
1	600	1	Sushi	3.99
2	155	1	Sushi	3.99
3	341	1	Sushi	3.99
4	20	1	Sushi	3.99
5	77	1	Sushi	3.99
6	100	1	Sushi	3.99
7	953	1	Sushi	3.99
8	504	1	Sushi	3.99
9	323	1	Sushi	3.99
10	922	1	Sushi	3.99
11	909	1	Sushi	3.99
12	602	1	Sushi	3.99
13	74	1	Sushi	3.99
14	703	1	Sushi	3.99
15	962	1	Sushi	3.99
16	968	1	Sushi	3.99
17	190	1	Sushi	3.99
18	163	1	Sushi	3.99
19	321	1	Sushi	3.99

Merging, Joining, and Concatenating

6. Right Joins

- Combine the two tables on a specific key. Only data on the intersection and right table will be taken. It is the same as *Left Join,* but in a different direction.
- Let's use the **right join** to get the customers' information.



Merging, Joining, and Concatenating

6. Right Joins

- First, we have to rename the column because in Customers dataset, the column name is ID.

```
df.head(3)
```

	Customer ID	Food ID	Food Item	Price
0	537	9	Donut	0.99
1	97	4	Quesadilla	4.25
2	658	1	Sushi	3.99

```
customers.head(3)
```

	ID	First Name	Last Name	Gender	Company	Occupation
0	1	Joseph	Perkins	Male	Dynazzy	Community Outreach Specialist
1	2	Jennifer	Alvarez	Female	DabZ	Senior Quality Engineer
2	3	Roger	Black	Male	Tagfeed	Account Executive

```
df.columns = ['ID', 'Food ID', 'Food Item', 'Price']
```

Merging, Joining, and Concatenating

6. Right Joins

- In our new DataFrame, the column name is Customer ID. We have standardized the name to ID.

```
In [240]: pd.merge(left=customers, right=df, on="ID", how="right").head()
```

Out[240]:

	ID	First Name	Last Name	Gender	Company	Occupation	Food ID	Food Item	Price
0	3	Roger	Black	Male	Tagfeed	Account Executive	2	Burrito	9.99
1	10	Steven	Ryan	Male	Twinder	Community Outreach Specialist	2	Burrito	9.99
2	20	Lisa	Rice	Female	Oloo	Programmer IV	1	Sushi	3.99
3	21	Albert	Burns	Male	Rhynoodle	Junior Executive	4	Quesadilla	4.25
4	21	Albert	Burns	Male	Rhynoodle	Junior Executive	4	Quesadilla	4.25

Merging, Joining, and Concatenating

7. left_on and right_on Parameters

- Let's say we want to combine two tables/dataset BUT both do not have the same column names.
- Worry NOT!
- We can specify different keys name for different tables using left_on and right_on parameters.

- In the previous technique, we have to rename the column first before combining which is a tedious process. Without renaming the column, we can still combine them by specifying the column names on both left and right.
- **right_on** : Column or index level names to join on in the right DataFrame.
- **left_on** : Use the index from the left DataFrame as the join key(s).

Merging, Joining, and Concatenating

7. left_on and right_on Parameters

```
In [92]: customers.head(3)
```

Out[92]:

	ID	First Name	Last Name	Gender	Company	Occupation
0	1	Joseph	Perkins	Male	Dynazzy	Community Outreach Specialist
1	2	Jennifer	Alvarez	Female	DabZ	Senior Quality Engineer
2	3	Roger	Black	Male	Tagfeed	Account Executive

```
In [91]: week1.head(3)
```

Out[91]:

	Customer ID	Food ID
0	537	9
1	97	4
2	658	1

```
In [95]: pd.merge(left=customers, right=week1, left_on="ID", right_on="Customer ID", sort=True).head()
```

Out[95]:

	ID	First Name	Last Name	Gender	Company	Occupation	Customer ID	Food ID
0	3	Roger	Black	Male	Tagfeed	Account Executive	3	2
1	10	Steven	Ryan	Male	Twinder	Community Outreach Specialist	10	2
2	20	Lisa	Rice	Female	Oloo	Programmer IV	20	1
3	21	Albert	Burns	Male	Rhynoodle	Junior Executive	21	4
4	21	Albert	Burns	Male	Rhynoodle	Junior Executive	21	4

Merging, Joining, and Concatenating

7. left_on and right_on Parameters

- ID column and Customer ID column are the same thing. After merging, Pandas will include both of them in the new DataFrame.
- We do not need to keep both as they are similar hence, we should drop any one of them.

```
In [96]: pd.merge(left=customers, right=week1, left_on="ID", right_on="Customer ID", sort=True).drop("ID", axis="columns")
```

Out[96]:

	First Name	Last Name	Gender	Company	Occupation	Customer ID	Food ID
0	Roger	Black	Male	Tagfeed	Account Executive	3	2
1	Steven	Ryan	Male	Twinder	Community Outreach Specialist	10	2
2	Lisa	Rice	Female	Oloo	Programmer IV	20	1
3	Albert	Burns	Male	Rhynoodle	Junior Executive	21	4
4	Albert	Burns	Male	Rhynoodle	Junior Executive	21	4
5	Michelle	Kelly	Female	Twitterbeat	Programmer Analyst I	26	9
6	Pamela	Hicks	Female	Ntags	Cost Accountant	30	2

Merging, Joining, and Concatenating

8. Merging two dataFrame based on Index

- In merge() method, there are right_index and left_index parameters. Set them to True to do merging using the particular index as the reference point.

```
In [97]: week1.head()
```

Out[97]:

	Customer ID	Food ID
0	537	9
1	97	4
2	658	1
3	202	2
4	155	9

```
In [99]: foods = pd.read_csv("data/Restaurant - Foods.csv", index_col="Food ID")
foods.head(3)
```

Out[99]:

Food ID	Food Item	Price
1	Sushi	3.99
2	Burrito	9.99
3	Taco	2.99

Merging, Joining, and Concatenating

8. Merging two dataFrame based on Index

- Here, we do not use the ***on*** parameter. Instead, we specify the **left_on** parameter and set the **right_index** parameter to True

```
In [100]: pd.merge(left=week1, right=foods, how="left", left_on="Food ID", right_index=True)
```

Out[100]:

	Customer ID	Food ID	Food Item	Price
0	537	9	Donut	0.99
1	97	4	Quesadilla	4.25
2	658	1	Sushi	3.99
3	202	2	Burrito	9.99
4	155	9	Donut	0.99
5	213	8	Salad	11.25
6	600	1	Sushi	3.99
7	503	5	Pizza	2.49
8	71	3	Taco	2.99
9	174	3	Taco	2.99
10	961	9	Donut	0.99
11	966	5	Pizza	2.49
12	641	4	Quesadilla	4.25
13	288	2	Burrito	9.99
14	149	4	Quesadilla	4.25

Merging, Joining, and Concatenating

9. **join()** method

- It works the same as merge() method.
- We can join/merge the table on the reference point(s) using this method as well.

```
In [102]: satisfaction = pd.read_csv("data/Restaurant - Week 1 Satisfaction.csv")
          satisfaction.head()
```

Out[102]:

	Satisfaction Rating
0	2
1	7
2	3
3	7
4	10

```
In [103]: week1.head(3)
```

Out[103]:

	Customer ID	Food ID
0	537	9
1	97	4
2	658	1

```
In [105]: week1.join(satisfaction).head()
```

Out[105]:

	Customer ID	Food ID	Satisfaction Rating
0	537	9	2
1	97	4	7
2	658	1	3
3	202	2	7
4	155	9	10

Merging, Joining, and Concatenating

9. **join()** method

- Using the merge() method, we can also get the same output. However, merge() method **requires more parameters.**

```
In [107]: pd.merge(left=week1, right=satisfaction, left_index=True, right_index=True)
```

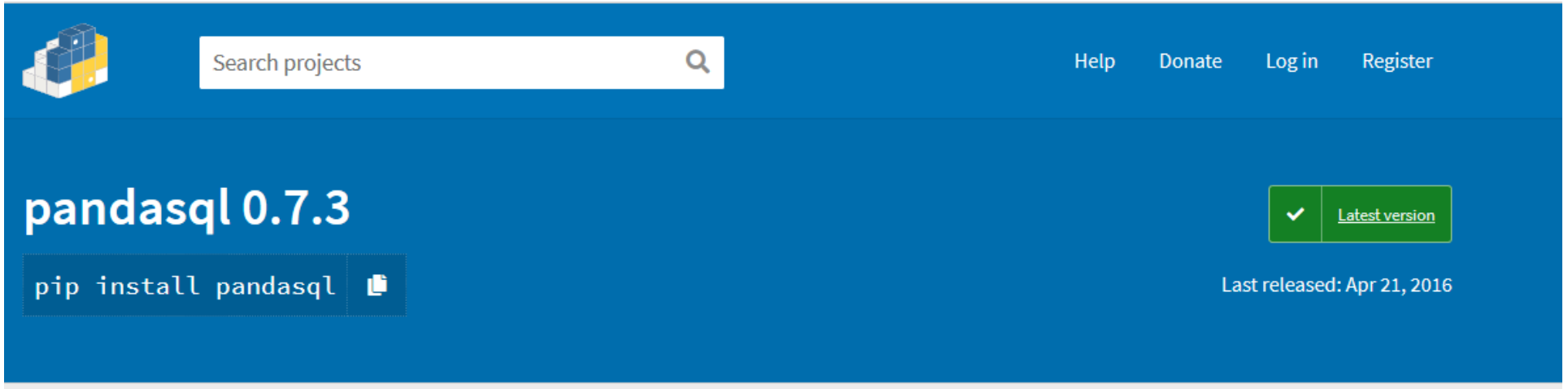
Out[107]:

	Customer ID	Food ID	Satisfaction Rating
0	537	9	2
1	97	4	7
2	658	1	3
3	202	2	7
4	155	9	10
5	213	8	3
6	600	1	2
7	503	5	5
8	71	3	10
9	174	3	7
10	961	9	7

Merging, Joining, and Concatenating

10. SQL command in Pandas

- To use the SQL command in Pandas, we need additional library called **PandasSQL**. https://pypi.org/project/pandasql/
- Installation command: *pip install pandasql*



Merging, Joining, and Concatenating

10. SQL command in Pandas

- Once the library installation is completed, you can import the library inside Jupyter Notebook.
- We use **sqldf() function** to query. We **pass SQL Query in string format.**
- For example, we want to select all columns in the food table. The query is as follows:
- **SELECT * from foods**

```
Successfully built pandasql
Installing collected packages: pandasql
Successfully installed pandasql-0.7.3
```

```
In [252]: import pandasql as ps
```

```
In [258]: ps.sqldf("SELECT * from foods").head(5)
```

Out[258]:

	Food ID	Food Item	Price
0	1	Sushi	3.99
1	2	Burrito	9.99
2	3	Taco	2.99
3	4	Quesadilla	4.25
4	5	Pizza	2.49

Merging, Joining, and Concatenating

10. Inner Join with SQL Query.

- Let's join Week1 and Week2 table using Customer ID as the primary key.
- We can combine the table using Customer ID as the key. However, the key name contains space which we need to **replace with underscore (_).**

```
In [256]: week1.columns = ["Customer_ID" , "Food_ID"]
          week2.columns = ["Customer_ID" , "Food_ID"]
```

```
In [254]: week1.head(3)
```

Out[254]:

	Customer ID	Food ID
0	537	9
1	97	4
2	658	1

```
In [255]: week2.head(3)
```

Out[255]:

	Customer ID	Food ID
0	688	10
1	813	7
2	495	10

Merging, Joining, and Concatenating

10. Inner Join with SQL Query.

- If we want to do the Inner join using SQL Command, the query is as follows:
- **SELECT * FROM week1 INNER JOIN week2 ON week1.Customer_ID = week2.Customer_ID**
- The query yields the same result as using Pandas Merge() function. If you are familiar with SQL commands, this library will help you a lot.

```
In [257]: ps.sqldf("SELECT * FROM week1  INNER JOIN week2 ON week1.Customer_ID = week2.Customer_ID").head(5)
```

Out[257]:

	Customer_ID	Food_ID	Customer_ID	Food_ID
0	537	9	537	5
1	155	9	155	3
2	503	5	503	8
3	503	5	503	9
4	155	1	155	3

DATA ANALYSIS WITH PYTHON

Module 8: Working with Dates and Times

Working with Dates and Times

Introduction

- Working with datetime type enables us to extract a lot of information. For examples, the quarters of a year (Q1,Q2,Q3,Q4)
- But first, let's explore on Python built-in date and time.
- **Datetime** is a built-in module.
- This module is not imported automatically for every python file to conserve/save memory.
- Hence, we need to import the module when we want to use it
- Datetime module has unique object, attributes, and method. We will learn the basics of Python Datetime before we learn Pandas Datetime format.

```
In [23]: import pandas as pd
         import datetime as dt
```

Working with Dates and Times

1. Review of Python's datetime Module

- An internal library that Python loads on demand. This is to conserve memory.
- The date is an object to store time data. It can be year, month, day and etc.
- **date() method**
  - Let's create a datetime object. The arguments are in order of year, month, and day.
  - *Hint* : Shift + Tab to read about the method
  - We can **directly call the attributes** from the object.

```
In [24]: someday = dt.date(2016,4,12)
         someday

Out[24]: datetime.date(2016, 4, 12)
```

```
In [25]: someday.day , someday.month , someday.year

Out[25]: (12, 4, 2016)
```

Working with Dates and Times

1. Review of Python's datetime Module

- **datetime() method**
  - It is the same as date() method but with additional time parameter. The **default values** for the time (hours and seconds) are (0000h) which is **midnight**.
  - Casting or changing the datetime object to string will give a readable date format.

```
In [26]: dt.datetime(2016,4,12)
Out[26]: datetime.datetime(2016, 4, 12, 0, 0)

In [27]: sometime = dt.datetime(2016,4,12, 8, 15)
         sometime
Out[27]: datetime.datetime(2016, 4, 12, 8, 15)

In [28]: str(someday)
Out[28]: '2016-04-12'
```

Working with Dates and Times

2. Pandas Timestamp Object

- Pandas version of Datetime
- If the user does not provide the specific time, the default values will be midnight (0000H)
- Pandas Datetime or Timestamp can do more analysis than Python Datetime module.
- **pd.Timestamp()**
  - argument can be Pandas Series or string
  - For string argument, Pandas can recognize the separators which are either dash (-) , comma (,) or slash (/)

```
In [29]: pd.Timestamp("2016-4-12")
Out[29]: Timestamp('2016-04-12 00:00:00')

In [30]: pd.Timestamp("4.12.2016")
Out[30]: Timestamp('2016-04-12 00:00:00')

In [31]: pd.Timestamp("2016/4/12 12:12")
Out[31]: Timestamp('2016-04-12 12:12:00')

In [32]: someday
Out[32]: datetime.date(2016, 4, 12)

In [33]: pd.Timestamp(someday)
Out[33]: Timestamp('2016-04-12 00:00:00')
```

Working with Dates and Times

3. The Pandas DateTimeIndex Object

- create dateTimeIndex object using DateTimeIndex() method.
- Change string type into datetime data type.

```
In [34]: dates = ["2015-10-5", "2016-5-10", "2019-1-20"]

In [35]: dtIndex = pd.DatetimeIndex(dates)
         dtIndex

Out[35]: DatetimeIndex(['2015-10-05', '2016-05-10', '2019-01-20'], dtype='datetime64[ns]', freq=None)
```

- We can also pass the Python Date/Datetime object to Pandas Datetime format.

```
In [36]: dates2 = [dt.date(2015,10,5), dt.date(2016,5,10), dt.date(2019,1,20)]
         pd.DatetimeIndex(dates2)

Out[36]: DatetimeIndex(['2015-10-05', '2016-05-10', '2019-01-20'], dtype='datetime64[ns]', freq=None)
```

Working with Dates and Times

3. The Pandas DateTimeIndex Object

- Let's create a Pandas Series with Datetime as index.

```
In [37]: values = [100,200,300]
         pd.Series(values, dtIndex)

Out[37]: 2015-10-05    100
         2016-05-10    200
         2019-01-20    300
         dtype: int64
```

Working with Dates and Times

4. pd.to_datetime() method

- Converts existing data object into Pandas Datetime object.
- to_datetime() can convert strings, Python Date object and Pandas Series.

```
In [38]: pd.to_datetime("2016-5-10")
Out[38]: Timestamp('2016-05-10 00:00:00')

In [39]: pd.to_datetime(dt.date(2015,10,5))
Out[39]: Timestamp('2015-10-05 00:00:00')

In [40]: pd.to_datetime(["2016-5-10", "2019-1-20", "July 12th 1998"])
Out[40]: DatetimeIndex(['2016-05-10', '2019-01-20', '1998-07-12'], dtype='datetime64[ns]', freq=None
```

- **Converting Pandas Series** into Pandas **Datetime object:**

```
In [41]: s = pd.Series(["2019-1-20", "July 12th 1998", "2016"])
         s
Out[41]: 0          2019-1-20
         1    July 12th 1998
         2              2016
         dtype: object

In [42]: pd.to_datetime(s)
Out[42]: 0   2019-01-20
         1   1998-07-12
         2   2016-01-01
         dtype: datetime64[ns]
```

Working with Dates and Times

4. pd.to_datetime() method

- If we pass the wrong string format to the method, it will prompt error.

```
In [43]: s = pd.Series(["2019-1-20", "July 12th 1998", "2016", "Hello World"])

In [44]: pd.to_datetime(s)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\Anaconda3\lib\site-packages\pandas\core\tools\datetimes.py in _convert_listli
ke(arg, box, format, name, tz)
    376             try:
--> 377                 values, tz = conversion.datetime_to_datetime64(arg)
    378                 return DatetimeIndex._simple_new(values, name=name, tz=
tz)
```

- However, we can fix the error by replacing it with **NaT** data, which means missing data. This can be done using errors parameter by setting it to **"coerce"**.

```
In [45]: pd.to_datetime(s, errors="coerce")

Out[45]: 0   2019-01-20
         1   1998-07-12
         2   2016-01-01
         3          NaT
         dtype: datetime64[ns]
```

Working with Dates and Times

4. pd.to_datetime() method

- UNIX time
  - Pandas can also read UNIX Time. Unix time (also known as POSIX time or UNIX Epoch time) is a system for describing a point in time.
  - It is the number of seconds that have elapsed since 00:00:00 Thursday, 1 January 1970, Coordinated Universal Time (UTC), minus leap seconds.

```
In [46]: pd.to_datetime(1564880922, unit="s")
Out[46]: Timestamp('2019-08-04 01:08:42')
```

Working with Dates and Times

5. Create Range of Dates with the pd.date_range() Method

- requires at least two parameters, start and end.
  - start : starting of day
  - end : upper bound for the day generated
- freq : how the internal is calculated. The default value is "D" which stands for Day. "2D" stands for 2 Days.
- For full list of freq : https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#timeseries-offset-aliases

Working with Dates and Times

5. Create Range of Dates with the pd.date_range() Method

- START + END + FREQ

```
In [47]: pd.date_range(start="2019-1-2", end="2019-1-10", freq="2D")

Out[47]: DatetimeIndex(['2019-01-02', '2019-01-04', '2019-01-06', '2019-01-08',
                        '2019-01-10'],
                       dtype='datetime64[ns]', freq='2D')
```

- We can also generate days on weekdays or business days. This can be done by specifying freq= "B“.

```
In [48]: pd.date_range(start="2019-1-2", end="2019-1-10", freq="B")

Out[48]: DatetimeIndex(['2019-01-02', '2019-01-03', '2019-01-04', '2019-01-07',
                        '2019-01-08', '2019-01-09', '2019-01-10'],
                       dtype='datetime64[ns]', freq='B')
```

Working with Dates and Times

5. Create Range of Dates with the pd.date_range() Method

- Let's generate the dates of all Fridays on February 2020.

```
In [49]: pd.date_range(start="1 Feb 2020", end="28 Feb 2020", freq="W-FRI")

Out[49]: DatetimeIndex(['2020-02-07', '2020-02-14', '2020-02-21', '2020-02-28'], dtype
         ='datetime64[ns]', freq='W-FRI')
```

- Generate ten days with 6 Hours interval.

```
In [50]: pd.date_range(start="1 Feb 2020", end="10 Feb 2020", freq="6H")

Out[50]: DatetimeIndex(['2020-02-01 00:00:00', '2020-02-01 06:00:00',
                        '2020-02-01 12:00:00', '2020-02-01 18:00:00',
                        '2020-02-02 00:00:00', '2020-02-02 06:00:00',
                        '2020-02-02 12:00:00', '2020-02-02 18:00:00',
                        '2020-02-03 00:00:00', '2020-02-03 06:00:00',
                        '2020-02-03 12:00:00', '2020-02-03 18:00:00',
```

Working with Dates and Times

5. Create Range of Dates with the pd.date_range() Method

- START + PERIODS + FREQ
- **periods parameter :** Number of dates we want to generate from starting date
- For example, if we want to generate **25 days** from 12 July 1998, we need to specify the **start parameter** as “12 July 1998” and set **periods parameter** to 25.

```
In [52]: pd.date_range(start="12 July 1998", periods=25, freq="D")

Out[52]: DatetimeIndex(['1998-07-12', '1998-07-13', '1998-07-14', '1998-07-15',
                        '1998-07-16', '1998-07-17', '1998-07-18', '1998-07-19',
                        '1998-07-20', '1998-07-21', '1998-07-22', '1998-07-23',
                        '1998-07-24', '1998-07-25', '1998-07-26', '1998-07-27',
                        '1998-07-28', '1998-07-29', '1998-07-30', '1998-07-31',
                        '1998-08-01', '1998-08-02', '1998-08-03', '1998-08-04',
                        '1998-08-05'],
                       dtype='datetime64[ns]', freq='D')
```

Working with Dates and Times

5. Create Range of Dates with the pd.date_range() Method

- We can also generate **25 Business days** (Weekdays) from 12 July 1998.
- Since 12 July is Sunday, the dates will start counting on 13 July.

```
In [53]: pd.date_range(start="12 July 1998", periods=25, freq="B")

Out[53]: DatetimeIndex(['1998-07-13', '1998-07-14', '1998-07-15', '1998-07-16',
                        '1998-07-17', '1998-07-20', '1998-07-21', '1998-07-22',
                        '1998-07-23', '1998-07-24', '1998-07-27', '1998-07-28',
                        '1998-07-29', '1998-07-30', '1998-07-31', '1998-08-03',
                        '1998-08-04', '1998-08-05', '1998-08-06', '1998-08-07',
                        '1998-08-10', '1998-08-11', '1998-08-12', '1998-08-13',
                        '1998-08-14'],
                       dtype='datetime64[ns]', freq='B')
```

Working with Dates and Times

5. Create Range of Dates with the pd.date_range() Method

- END + PERIODS + FREQ
- We can also specify the end dates and the number of periods we want, then it will generate a list of dates which will end before the *end date*
- For example, the code below will generate 10 dates with an interval of 1 day. The end date is 31 DEC 2019. Hence, the calculated starting date is 22 DEC 2019

```
In [57]: pd.date_range(end="31 Dec 2019", periods=10, freq="D")

Out[57]: DatetimeIndex(['2019-12-22', '2019-12-23', '2019-12-24', '2019-12-25',
                        '2019-12-26', '2019-12-27', '2019-12-28', '2019-12-29',
                        '2019-12-30', '2019-12-31'],
                       dtype='datetime64[ns]', freq='D')
```

Working with Dates and Times

5. Create Range of Dates with the pd.date_range() Method

- **Freq="MS"** means start of each month.
- Notice the last date is 1 Dec 2019, because our end date is 31 Dec 2019. Hence, dates after the end date will not be generated.

```
In [58]: pd.date_range(end="31 Dec 2019", periods=10, freq="MS")

Out[58]: DatetimeIndex(['2019-03-01', '2019-04-01', '2019-05-01', '2019-06-01',
                        '2019-07-01', '2019-08-01', '2019-09-01', '2019-10-01',
                        '2019-11-01', '2019-12-01'],
                       dtype='datetime64[ns]', freq='MS')
```

Working with Dates and Times

6 .dt Accessor

- In the previous chapter, we learned how to change a columns' data type from string to datetime. This is to enable us to do more operations on the date.
- Let’s generate a list of dates and pass it into Pandas Series.

```
In [59]: list_dates = pd.date_range(start="1-1-2019", end="10-1-2019", freq="B")

In [60]: s = pd.Series(list_dates)
         s.head()

Out[60]: 0   2019-01-01
         1   2019-01-02
         2   2019-01-03
         3   2019-01-04
         4   2019-01-07
         dtype: datetime64[ns]

In [61]: len(s)

Out[61]: 196
```

Working with Dates and Times

6 .dt Accessor

- Now we have a list of dates in Pandas Series. We need to use **.dt** to access some of the attributes
- **s.dt.day** will return the days from the Series. Since the Series is already in datetime type, we do not need to do additional operations.

- s.dt.is_quater_start will return a list of Booleans. It will return TRUE if the date is the first day of a quarter.

```
In [62]: s.dt.day.head()

Out[62]: 0    1
         1    2
         2    3
         3    4
         4    7
         dtype: int64
```

```
In [63]: mask = s.dt.is_quarter_start
         s[mask]

Out[63]: 0      2019-01-01
         64     2019-04-01
         129    2019-07-01
         195    2019-10-01
         dtype: datetime64[ns]
```

Working with Dates and Times

6 .dt Accessor

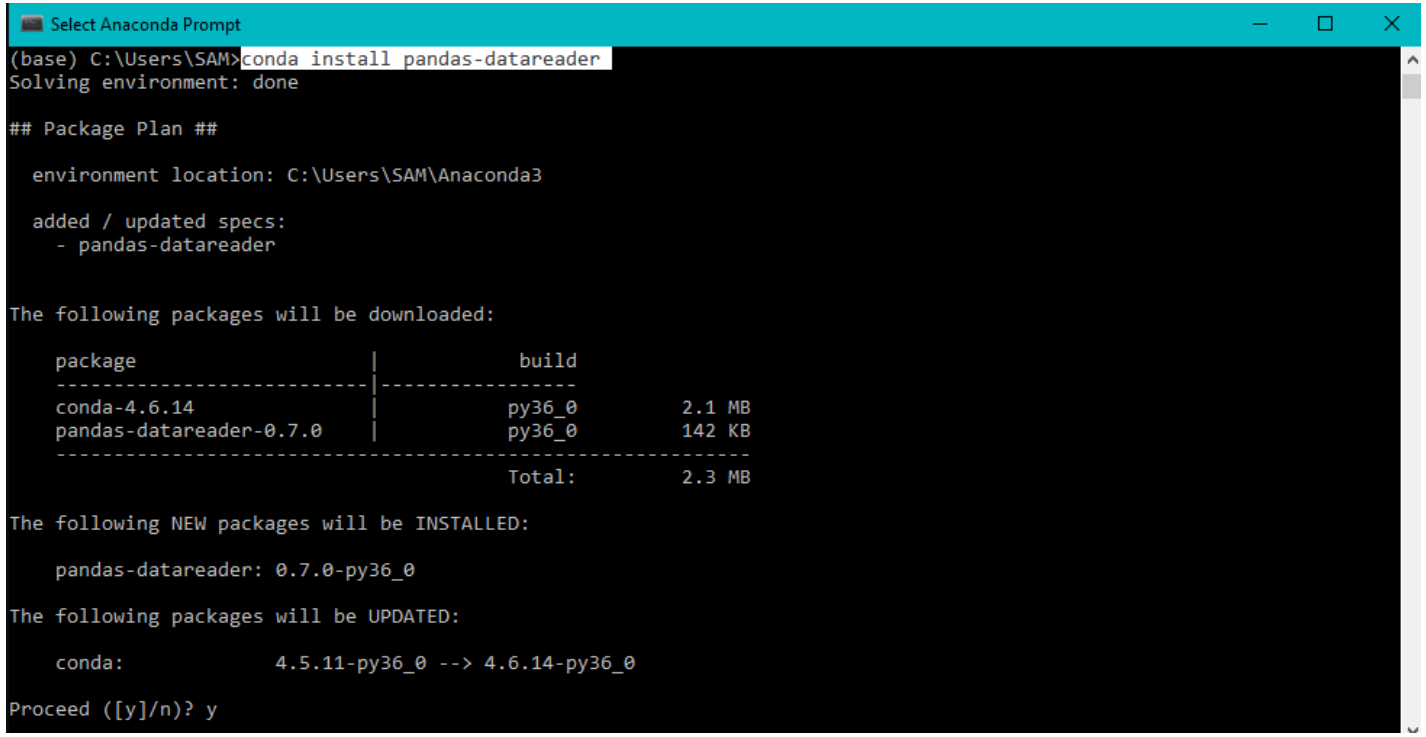
- Generating Business days/Weekday, will gives us the days from Monday to Friday.

```
In [64]: s.dt.day_name()

Out[64]: 0       Tuesday
         1     Wednesday
         2      Thursday
         3        Friday
         4        Monday
         5       Tuesday
         6     Wednesday
         7      Thursday
```

Working with Dates and Times

7. Install Pandas-datareader library

- pandas_datareader provides a consistent, simple API for you to collect data from these platforms. We can get stock prices for many companies.
- Open Anaconda Command Prompt.
- Run command - **conda install pandas-datareader.**
- Wait until it is Done!



Working with Dates and Times

8. Import Financial Datasets with Pandas Data Reader Library

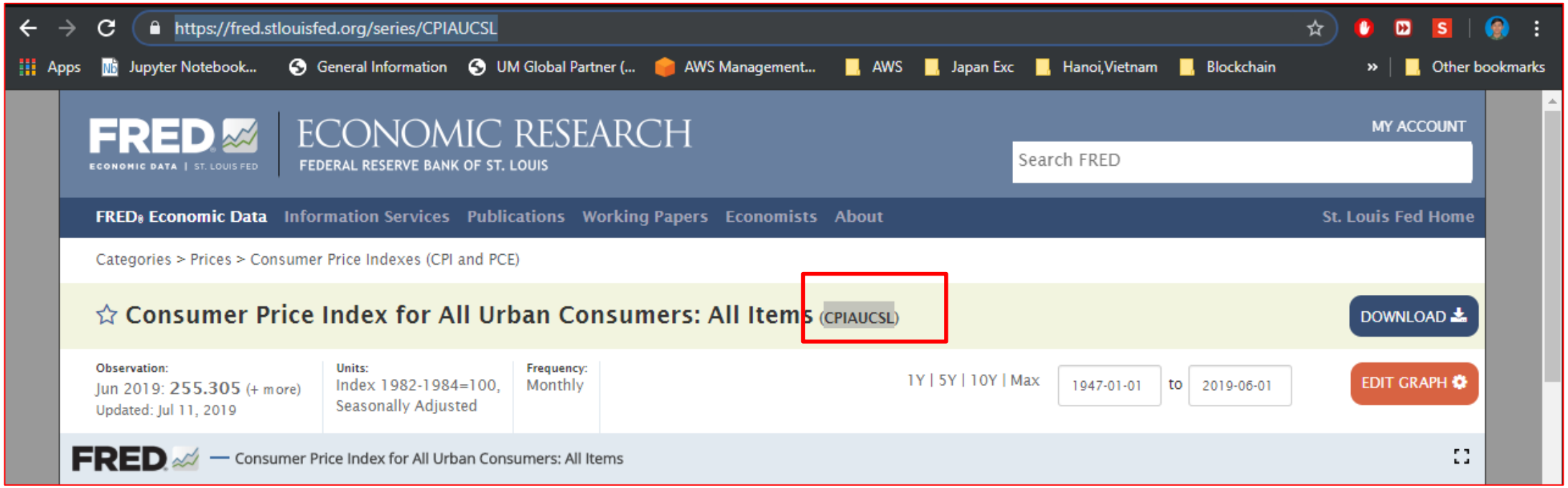
- Unfortunately, Google finance and Morningstar have been deprecated due to massive changes in their API and no stable replacement.
- Full documentation : https://pydata.github.io/pandas-datareader/stable
- First, we import data from the **pandas_datareader** module.

```
In [65]: import pandas as pd
         import datetime as dt
         from pandas_datareader import data
```

Working with Dates and Times

8. Import Financial Datasets with Pandas Data Reader Library

- FRED (Federal Reserve Economic Data)
- source : https://pydata.github.io/pandas-datareader/stable/remote_data.html#fred
- From FRED Official Website: https://fred.stlouisfed.org/series/CPIAUCSL



Working with Dates and Times

8. Import Financial Datasets with Pandas Data Reader Library

- Use the code shown to retrieve the data using Pandas DataReader.
- Notice that “CPIAUCSL” is taken from FRED Website. Since each dataset has its unique key value.

```
In [66]: data.get_data_fred('CPIAUCSL')
```

Out[66]:

DATE	CPIAUCSL
2010-01-01	217.488
2010-02-01	217.281
2010-03-01	217.353
2010-04-01	217.403
2010-05-01	217.290
2010-06-01	217.199

Working with Dates and Times

8. Import Financial Datasets with Pandas Data Reader Library

- FRED Source: https://fred.stlouisfed.org/series/GDP


```
In [70]: from pandas_datareader import data
start="1990-1-2"
end="2019-1-10"
gdp = data.DataReader('GDP', 'fred', start, end)
gdp
```

Out[70]:

DATE	GDP
1990-04-01	5960.028
1990-07-01	6015.116
1990-10-01	6004.733
1991-01-01	6035.178
1991-04-01	6126.862
1991-07-01	6205.937
1991-10-01	6264.540
1992-01-01	6363.102
1992-04-01	6470.763
1992-07-01	6566.641
1992-10-01	6680.803

Working with Dates and Times

8. Import Financial Datasets with Pandas Data Reader Library

- **Fama/French**
- Another good source of Data is Fama/French
- Fama and French Three-Factor Model (or the Fama French Model for short) is an asset pricing model developed in 1992 that expands on the capital asset pricing model (CAPM) source : https://www.investopedia.com/terms/f/famaandfrenchthreefactormodel.asp
- source : https://pydata.github.io/pandas-datareader/stable/remote_data.html#fama-french

```
In [71]: from pandas_datareader import data
         ds = data.DataReader('5_Industry_Portfolios', 'famafrench')
         ds
```

Out[71]: {0:

Date	Cnsmr	Manuf	HiTec	Hlth	Other
2010-01	-1.98	-4.02	-7.63	0.00	-1.65
2010-02	3.86	3.23	4.37	0.38	3.38
2010-03	6.33	4.74	6.92	3.61	8.43
2010-04	1.53	3.31	2.50	-2.22	2.15
2010-05	-5.71	-8.88	-7.31	-8.01	-8.82
2010-06	-6.47	-5.19	-5.77	-1.63	-7.65
2010-07	6.44	8.48	7.98	2.15	7.48
2010-08	-3.32	-3.79	-5.54	-1.66	-7.55
2010-09	9.85	8.83	11.41	9.01	8.82
2010-10	3.62	4.33	6.04	2.00	2.60
2010-11	2.84	2.46	-1.04	-3.34	0.52
2010-12	4.12	7.27	6.18	5.40	9.73
2011-01	-1.31	4.25	3.10	-0.71	1.97
2011-02	2.91	4.88	3.40	3.33	2.66

Working with Dates and Times

8. Import Financial Datasets with Pandas Data Reader Library

```
In [72]: ds.keys()

Out[72]: dict_keys([0, 1, 2, 3, 4, 5, 6, 7, 'DESCR'])

In [73]: ds["DESCR"]

Out[73]: '5 Industry Portfolios\n----------------------\n\nThis file was created by CMPT_
         IND_RETS using the 201906 CRSP database. It contains value- and equal-weighted
         returns for 5 industry portfolios. The portfolios are constructed at the end of
         June. The annual returns are from January to December. Missing data are indicat
         ed by -99.99 or -999. Copyright 2019 Kenneth R. French\n\n  0 : Average Value W
         eighted Returns -- Monthly (114 rows x 5 cols)\n  1 : Average Equal Weighted Re
         turns -- Monthly (114 rows x 5 cols)\n  2 : Average Value Weighted Returns -- A
         nnual (9 rows x 5 cols)\n  3 : Average Equal Weighted Returns -- Annual (9 rows
         x 5 cols)\n  4 : Number of Firms in Portfolios (114 rows x 5 cols)\n  5 : Avera
         ge Firm Size (114 rows x 5 cols)\n  6 : Sum of BE / Sum of ME (9 rows x 5 cols)
         \n  7 : Value-Weighted Average of BE/ME (9 rows x 5 cols)'
```

Working with Dates and Times

9. Selecting DataFrame with DataTimeIndex

- **Fama/French**
- Another good source of Data is Fama/French
- Fama and French Three-Factor Model (or the Fama French Model for short) is an asset pricing model developed in 1992 that expands on the capital asset pricing model (CAPM) source : https://www.investopedia.com/terms/f/famaandfrenchthreefactormodel.asp
- source : https://pydata.github.io/pandas-datareader/stable/remote_data.html#fama-french

```
In [71]: from pandas_datareader import data
         ds = data.DataReader('5_Industry_Portfolios', 'famafrench')
         ds
```

Out[71]: {0:

Date	Cnsmr	Manuf	HiTec	Hlth	Other
2010-01	-1.98	-4.02	-7.63	0.00	-1.65
2010-02	3.86	3.23	4.37	0.38	3.38
2010-03	6.33	4.74	6.92	3.61	8.43
2010-04	1.53	3.31	2.50	-2.22	2.15
2010-05	-5.71	-8.88	-7.31	-8.01	-8.82
2010-06	-6.47	-5.19	-5.77	-1.63	-7.65
2010-07	6.44	8.48	7.98	2.15	7.48
2010-08	-3.32	-3.79	-5.54	-1.66	-7.55
2010-09	9.85	8.83	11.41	9.01	8.82
2010-10	3.62	4.33	6.04	2.00	2.60
2010-11	2.84	2.46	-1.04	-3.34	0.52
2010-12	4.12	7.27	6.18	5.40	9.73
2011-01	-1.31	4.25	3.10	-0.71	1.97
2011-02	2.91	4.88	3.40	3.33	2.66

DATA ANALYSIS WITH PYTHON

Module 9: Input and Output

Input and Output

1. Creating a random 3D dataset

- Link : https://data.cityofnewyork.us/City-Government/NYC-Civil-Service-Titles/nzjr-3966
- pd.read_csv() can also read CSV links from websites. The link above was used in the figure.

```
In [62]: import pandas as pd

In [63]: URL = r"https://data.cityofnewyork.us/api/views/nzjr-3966/files/4b888c87-1952-4ae6-b820-df0e6576db79?download=true&filename=Data_
         civil = pd.read_excel(URL)
         civil.head()
```

Out[63]:

	Note: Hover over each column header for an explanation of each field.	Unnamed: 1	Unnamed: 2	Unnamed: 3	Unnamed: 4	Unnamed: 5	Unnamed: 6	Unnamed: 7	Unnamed: 8	Unnamed: 9	Unnamed: 10
0	Order	Field Name	Longform Name	Description	Geocoded	Required	Data Type	Expected/Allowed Values	Last Modified Date	No Longer In Use	Notes
1	1	Title Code	NaN	Civil Service Title Code	False	True	Plain Text	NaN	NaN	NaN	NaN
2	2	Title Description	NaN	Name/description of the title	False	True	Plain Text	NaN	NaN	NaN	NaN
3	3	Standard Hours	NaN	Standard weekly hours for the title	False	True	Number	NaN	NaN	NaN	NaN
4	4	Assignment Level	NaN	The assignment level within the title	False	True	Number	NaN	NaN	NaN	NaN

Input and Output

1. Creating a random 3D dataset

- Link : https://data.cityofnewyork.us/api/views/nzjr-3966/rows.csv?accessType=DOWNLOAD

```
In [64]: URL =r"https://data.cityofnewyork.us/api/views/nzjr-3966/rows.csv?accessType=DOWNLOAD"
civil_service = pd.read_csv(URL)
civil_service.head()
```

Out[64]:

	Title Code	Title Description	Standard Hours	Assignment Level	Union Code	Union Description	Bargaining Unit Short Name	Bargaining Unit Description	Minimum Salary Rate	Maximum Salary Rate
0	00031	HCPPA	37.0	00	23	PRINCIPAL ADMINISTRATIVE ASSOC	CWA	Communication Wrkrs of America	42451.0	79294.0
1	00136	BEAUTICIAN	37.5	00	124	L420,L2507,L3627 DC37 HOSPITAL	DC37	District Council 37	39298.0	46084.0
2	00486	HOME HEALTH AIDE	37.5	00	127	INSTITUTIONAL SERVICE TITLES D	DC37	District Council 37	39175.0	45764.0
3	03647	SPECIAL ASSISTANT TO THE BOROU	35.0	00	775	MGRL TEMP - PROP NC	N/U	Non-Union	61457.0	244820.0
4	03927	ASSISTANT SYSTEMS ANALYST (HHC	35.0	00	129	ACCOUNTING AND EDP TITLES DC37	DC37	District Council 37	21771.0	40314.0

Input and Output

2. Quick Object conversion

- Pandas also has the **list()** method to convert the Pandas Series into Python List.

```
In [84]: civil_service["Title Description"].tolist()

Out[84]: ['HCPPA',
          'BEAUTICIAN',
          'HOME HEALTH AIDE',
          'SPECIAL ASSISTANT TO THE BOROU',
          'ASSISTANT SYSTEMS ANALYST (HHC',
          'ASSISTANT TO THE PRESIDENT (QN',
          'DIRECTOR OF INTERGOVERNMENTAL',
          'FOOD SERVICE MANAGER',
          'ASSISTANT TO THE PRESIDENT (ST',
          'STUDENT LEGAL SPECIALIST',
          'EXECUTIVE DIRECTOR OF ADMINIST',
          'DIRECTOR (PLANT OPERATIONS SER',
          'ASSISTANT TO THE PRESIDENT (BR',
          'SECRETARY TO ASSISTANT TO PRES',
          'SECRETARY TO THE EXCUTIVE ASSI',
          'ASSOCIATE DIRECTOR-OPERATIONAL',
          'RESEARCH AND LIAISON COORDINAT',
          'RESEARCH LIAISON ADN GOVERNMEN',
          'DIRECTOR OF REGIONAL JOINT INT',
```

Input and Output

2. Quick Object conversion

- We can convert Pandas Series to DataFrame by using **to_frame()**.

```
In [85]: civil_service["Title Description"].to_frame().head()
```

Out[85]:

	Title Description
0	HCPPA
1	BEAUTICIAN
2	HOME HEALTH AIDE
3	SPECIAL ASSISTANT TO THE BOROU
4	ASSISTANT SYSTEMS ANALYST (HHC

Input and Output

2. Quick Object conversion

- Combine every Title Description into one string and separate by comma.

```
In [86]: ", ".join(civil_service["Title Description"])
```

```
Out[86]: "HCPPA, BEAUTICIAN, HOME HEALTH AIDE, SPECIAL ASSISTANT TO THE BOROU, ASSISTANT SYSTEMS ANALYST (HHC, ASSISTANT TO THE PRESID
ENT (QN, DIRECTOR OF INTERGOVERNMENTAL, FOOD SERVICE MANAGER, ASSISTANT TO THE PRESIDENT (ST, STUDENT LEGAL SPECIALIST, EXECU
TIVE DIRECTOR OF ADMINIST, DIRECTOR (PLANT OPERATIONS SER, ASSISTANT TO THE PRESIDENT (BR, SECRETARY TO ASSISTANT TO PRES, SE
CRETARY TO THE EXCUTIVE ASSI, ASSOCIATE DIRECTOR-OPERATIONAL, RESEARCH AND LIAISON COORDINAT, RESEARCH LIAISON ADN GOVERNMEN,
DIRECTOR OF REGIONAL JOINT INT, DIRECTOR BOROUGH PRESIDENT'S O, ASSISTANT TO THE PRESIDENT(BOR, ASSISTANT TO THE PRESIDENT(BO
R, FISCAL AND POLICY ANALYST (BP, CHAUFFEUR-ATTENDANT (BKLYN BOR, DEPUTY OPERATION SUPPORT MANAG, RESEARCH AND LIAISON SPECIA
LIS, CHAUFFEUR-ATTENDANT (BORO PRES, CHAUFFEUR-ATTENDANT (BORO PRES, RESEARCH PROJECTS COORDINATOR, ASSISTANT TO THE DEPUTY M
AYOR, RESEARCH PROJECTS COOR(MA)-MGR, SPECIAL ASST TO THE DIRECTOR O, PARK BOROUGH COMMISSIONER (PAR, SENIOR RACKETS INVESTIG
ATOR (B, SUPERVISING RACKETS INVESTIGAT, SPECIAL ASSISTANT TO DA (QUEEN, PRINCIPAL ACCOUNTANT INVESTIGA, EMPLOYEE HEALTH BENE
FITS PROGR, CHIEF ADMINISTRATOR OF IMPARTI, DIRECTOR, DATA PROCESSING COOR, DIRECTOR OF BUILDING MANAGEMEN, STATISTICAL SECRE
TARY (OMB), STATISTICAL SECRETARY (OMB), DEPUTY COMMISSIONER (TAXI AND, GENERAL COUNSEL (TAXI & LIMOUS, LEGISLATIVE AIDE (OFF
ICE OF TH, SECRETARY (OFFICE OF THE MAYOR, DEPUTY BOROUGH COMMISSIONER (P, DEPUTY DIRECTOR FOR MAINTENANC, DIRECTOR OF ENFORC
EMENT (TAXI, EXECUTIVE ADMINISTRATOR OF GRA, ASSOCIATE ARTS PROGRAM SPECIAL, EXECUTIVE ASSISTANT TO THE CIT, DIRECTOR, OFFICE
OF OPERATIONS, ASST EXECUTIVE DIRECTOR FOR PR, ASST DIR OF INTERGVNMENTAL REL, DEPUTY ASSISTANT DIRECTOR (CIV, PROJECT PLANNE
R (MA), SR PROJECT PLANNER (MA), ASSISTANT COMMISSIONER(ADMINIS, DIR OF ADMINISTRATION, WORKER', PRINCIPAL ASSISTANT TO DEPUT
Y, EXECUTIVE ASSISTANT-MIDTOWN EN, SPECIAL ADVISOR TO CITY CLERK, SECRETARY (OFFICE OF BORO PRES, ASST PROJECT PLANNER (OFFIC
E O, PROJECT PLANNER (OFFICE OF BOR, CONFIDENTIAL ASSISTANT TO BORO, RECREATION SPECIALIST (DEPT OF, BUDGET ANALYST (OMB), BU
DGET ANALYST (OMB), BUDGET ANALYST (OMB)-MANAGERIA, CORRECTIONAL STANDARDS REVIEW, CORRECTIONAL STANDARDS REVIEW, DIRECTOR OF
CRISIS ASSISTANCE, ASSISTANT LEGISLATIVE REPRESEN, CHAUFFEUR-ATTENDANT (BORO PRES, PUBLIC RELATIONS OFFICER (BORO, SUPERVISOR
```

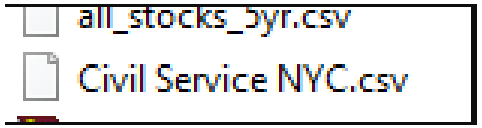
Input and Output

3. Export dataFrame into CSV file

- We use **pd.to_csv()** method to export dataFrame. In this method, there are a few **parameters** that will help us a lot:
  - **path** : The filename we have to save. Always remember to include **.csv** at the end of the string.
  - **columns** : Select column names to export.
  - **index** : By default, it is set to False. If True, the index will be removed in the CSV file.
  - **encoding** : "utf-8“ is the most popular character encoding. If you encounter an error when encoding, try to set this value.

```
In [89]: civil_service.to_csv("Civil Service NYC.csv",index=False, columns=["Title Description","Standard Hours", "Maximum Salary Rate"],)
```

- There will be no output when running this code. However, we can see a file is generated.



Input and Output

4. Install XLRD Library

- This library is used to read an Excel file using Pandas.
- It is included in the Anaconda packages

```
(base) C:\Users\SAM>conda install xlrd
Collecting package metadata: done
```

Input and Output

5. Read Excel Files into Pandas

- If the Excel file contains only one spreadsheet, Pandas will create the first spreadsheet by default.
- When you have multiple spreadsheets, we need to specify which sheet we want to read.

```
In [90]: pd.read_excel("data/Data - Single Worksheet.xlsx")
```

Out[90]:

	First Name	Last Name	City	Gender
0	Brandon	James	Miami	M
1	Sean	Hawkins	Denver	M
2	Judy	Day	Los Angeles	F
3	Ashley	Ruiz	San Francisco	F
4	Stephanie	Gomez	Portland	F

- Multiple spreadsheets

```
In [91]: pd.read_excel("data/Data - Multiple Worksheets.xlsx", sheet_name=0)
```

Out[91]:

	First Name	Last Name	City	Gender
0	Brandon	James	Miami	M
1	Sean	Hawkins	Denver	M
2	Judy	Day	Los Angeles	F
3	Ashley	Ruiz	San Francisco	F
4	Stephanie	Gomez	Portland	F

```
In [92]: pd.read_excel("data/Data - Multiple Worksheets.xlsx", sheet_name=1)
```

Out[92]:

	First Name	Last Name	City	Gender
0	Parker	Power	Raleigh	F
1	Preston	Prescott	Philadelphia	F
2	Ronaldo	Donaldo	Bangor	M
3	Megan	Stiller	San Francisco	M
4	Bustin	Jieber	Austin	F

Input and Output

5. Read Excel Files into Pandas

- Multiple spreadsheets

```
In [91]: pd.read_excel("data/Data - Multiple Worksheets.xlsx", sheet_name=0)
```

Out[91]:

	First Name	Last Name	City	Gender
0	Brandon	James	Miami	M
1	Sean	Hawkins	Denver	M
2	Judy	Day	Los Angeles	F
3	Ashley	Ruiz	San Francisco	F
4	Stephanie	Gomez	Portland	F

```
In [92]: pd.read_excel("data/Data - Multiple Worksheets.xlsx", sheet_name=1)
```

Out[92]:

	First Name	Last Name	City	Gender
0	Parker	Power	Raleigh	F
1	Preston	Prescott	Philadelphia	F
2	Ronaldo	Donaldo	Bangor	M
3	Megan	Stiller	San Francisco	M
4	Bustin	Jieber	Austin	F

- We can also choose the sheets based on the sheet's name. For example, the second sheet is "Data 2"

```
In [93]: pd.read_excel("data/Data - Multiple Worksheets.xlsx", sheet_name="Data 2")
```

Out[93]:

	First Name	Last Name	City	Gender
0	Parker	Power	Raleigh	F
1	Preston	Prescott	Philadelphia	F
2	Ronaldo	Donaldo	Bangor	M
3	Megan	Stiller	San Francisco	M
4	Bustin	Jieber	Austin	F

Input and Output

6. Export Excel file

- Create Excel Writer object.

```
civilExcel = pd.ExcelWriter("Civil Service.xlsx")
```

- Add the DataFrame into the ExcelWriter object.

```
title = civil_service[["Title Code", "Title Description"]]
union = civil_service[["Union Code","Union Description"]]
```

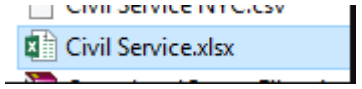
```
title.to_excel(civilExcel, sheet_name="Title")
```

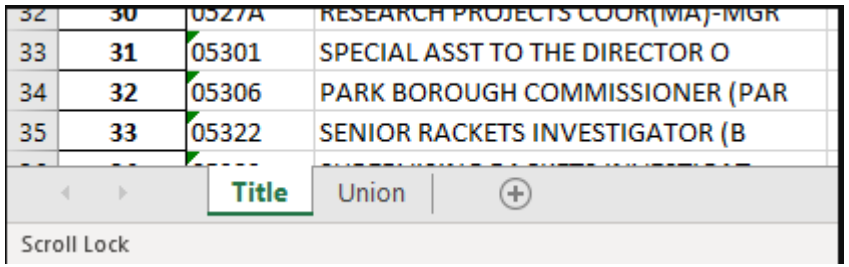
```
union.to_excel(civilExcel, sheet_name="Union")
```

- Save the Excel Writer object to **.xlsx** file.

```
civilExcel.save()
```

- Search for the file in the same folder.





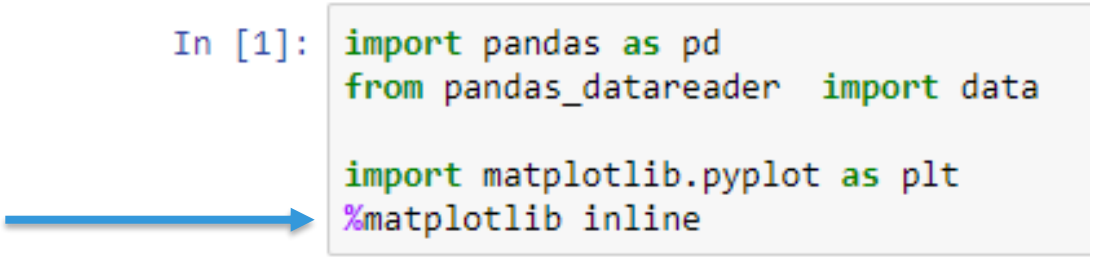
DATA ANALYSIS WITH PYTHON

Module 10: Visualization

Visualization

1. plot() method

- Takes the data and displays it visually (graph, chart).
- We can understand the data better with visual aid. For instance, the trend of data, overview of the data, preferences in classes.
- We will be working with matplotlib module.
- **%matplotlib inline**: renders the visual picture in Jupyter Notebook. Without this line, the matplotlib library will prompt the visual in a new window.



Visualization

1. plot() method

In [31]:
```
all_companies = pd.read_csv("data/all_stocks_5yr.csv", index_col=["date"], parse_dates=["date"])
all_companies.head()
```

Out[31]:

date	open	high	low	close	volume	Name
2013-02-08	15.07	15.12	14.63	14.75	8407500	AAL
2013-02-11	14.89	15.01	14.26	14.46	8882000	AAL
2013-02-12	14.45	14.51	14.10	14.27	8126000	AAL
2013-02-13	14.30	14.94	14.25	14.66	10259500	AAL
2013-02-14	14.94	14.96	13.16	13.99	31879900	AAL

In [32]:
```
mask = all_companies.Name == "FB"
FB = all_companies[mask].copy()
```

In [33]:
```
FB.drop("Name", axis = "columns", inplace=True)
```

Visualization

1. plot() method

- When applying the plot() method on dataFrame, all columns are plotted on the graph.
- **index == X-axis, columns == Y-axis.**
- This graph is hard to understand. We can only see the Volume data being plotted. Since Volume values are a lot bigger than the other columns, the line plotted for open, high, low and close values cannot be seen.

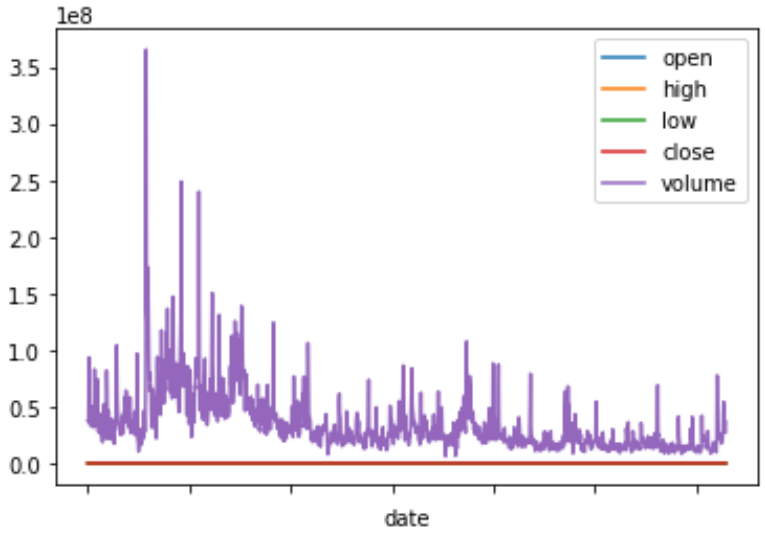
```
In [34]: FB.head(3)
```

Out[34]:

date	open	high	low	close	volume
2013-02-08	28.89	29.17	28.51	28.545	37662614
2013-02-11	28.61	28.68	28.04	28.260	36979533
2013-02-12	27.67	28.16	27.10	27.370	93417215

```
In [22]: FB.plot()
```

```
Out[22]: <matplotlib.axes._subplots.AxesSubplot at 0x246cf53bc18>
```

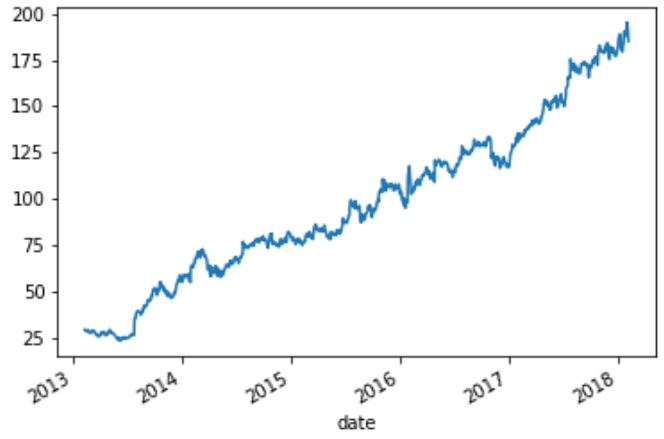


Visualization

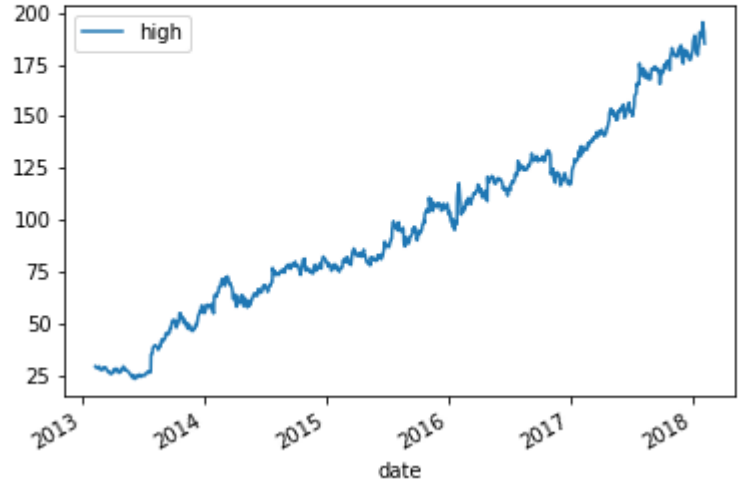
1. plot() method

- In the **plot()** method, it has the y parameter. This parameter allows us to choose which columns to be the y-axis.

```
In [40]: FB["high"].plot()
Out[40]: <matplotlib.axes._subplots.AxesSubplot at 0x246cf986668>
```



```
In [39]: FB.plot(y="high")
Out[39]: <matplotlib.axes._subplots.AxesSubplot at 0x246cfe1c3c8>
```

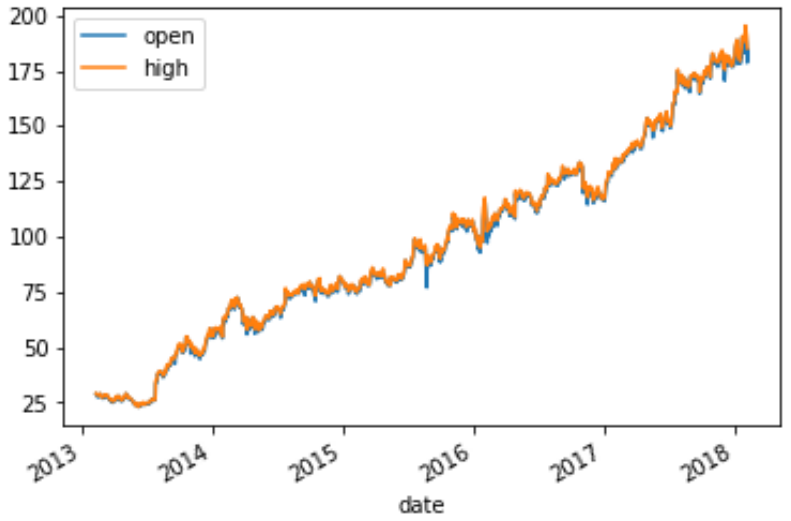


Visualization

1. plot() method

- When plotting two columns in one graph, we extract the dataFrame from the original dataFrame.
- Then, we use the **plot() method** to create a graph using the dataFrame as the data source.

```
In [44]: FB[["open", "high"]].plot()
Out[44]: <matplotlib.axes._subplots.AxesSubplot at 0x246cfb1a3c8>
```



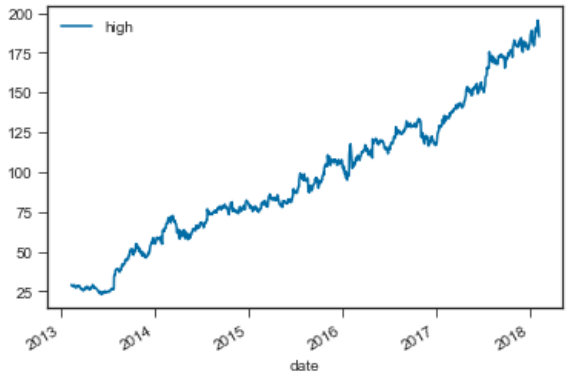
Visualization

2. Modifying Aesthetics

- We can stylize the graph to have different looks. Choose a style from the list of styles available.

```
In [45]: plt.style.available
Out[45]: ['bmh',
          'classic',
          'dark_background',
          'fast',
          'fivethirtyeight',
          'ggplot',
          'grayscale',
          'seaborn-bright',
          'seaborn-colorblind',
          'seaborn-dark-palette',
          'seaborn-dark',
          'seaborn-darkgrid',
          'seaborn-deep',
          'seaborn-muted',
          'seaborn-notebook',
          'seaborn-paper',
```

```
In [50]: plt.style.use("tableau-colorblind10")
         FB.plot(y="high")
Out[50]: <matplotlib.axes._subplots.AxesSubplot at 0x246cfd0f860>
```




```
In [51]: plt.style.use("ggplot")
         FB.plot(y="high")
Out[51]: <matplotlib.axes._subplots.AxesSubplot at 0x246cfd7a0f0>
```

Visualization

3. Bar Charts

- A bar chart is a chart or graph that presents categorical data.
- **Position** holds values in categories (PG, SF,C, etc).
- From Position column, we can visualize each position value in a graph.
- We then use the **value_counts()** method to find unique values in each category.

```
nba = pd.read_csv("data/nba.csv")
nba.head()
```

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN
3	R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1148640.0
4	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN	5000000.0

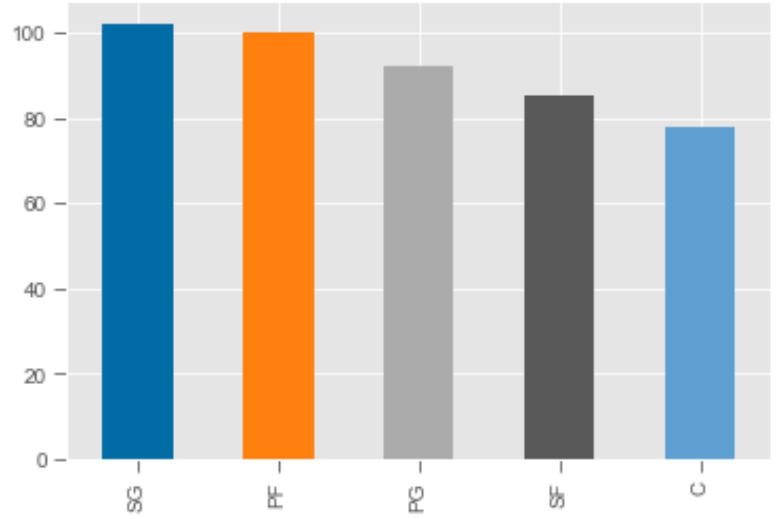
```
In [66]: nba["Position"].value_counts()

Out[66]: SG    102
         PF    100
         PG     92
         SF     85
         C      78
         Name: Position, dtype: int64
```

Visualization

3. Bar Charts

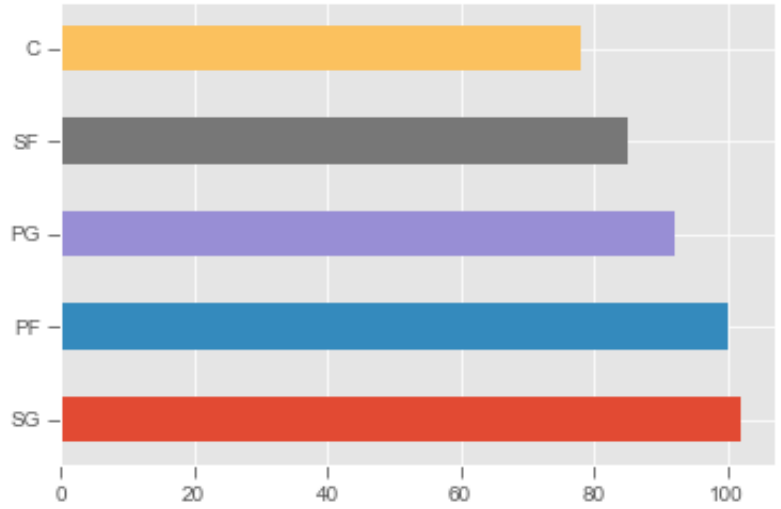
- **kind** indicates the type of graph to display. By default, it is "line" which stands for line graph.

```
In [69]: plt.style.use("tableau-colorblind10")
         nba["Position"].value_counts().plot(kind="bar")

Out[69]: <matplotlib.axes._subplots.AxesSubplot at 0x246d04faa
```



- **kind = "barh"** stands for horizontal bar graph. This will swap the x-axis and y-axis

```
In [65]: nba["Position"].value_counts().plot(kind="barh")

Out[65]: <matplotlib.axes._subplots.AxesSubplot at 0x246d04160b8>
```

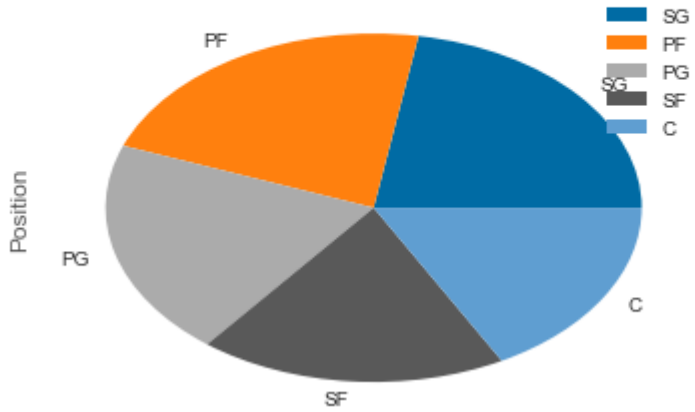


Visualization

4. Pie Chart

- **kind = "pie"** represents a pie chart graph.
- Notice the additional parameter, legend. This parameter shows which class the colors represent.

```
In [71]: nba["Position"].value_counts().plot(kind="pie", legend = True)
Out[71]: <matplotlib.axes._subplots.AxesSubplot at 0x246d05b04a8>
```

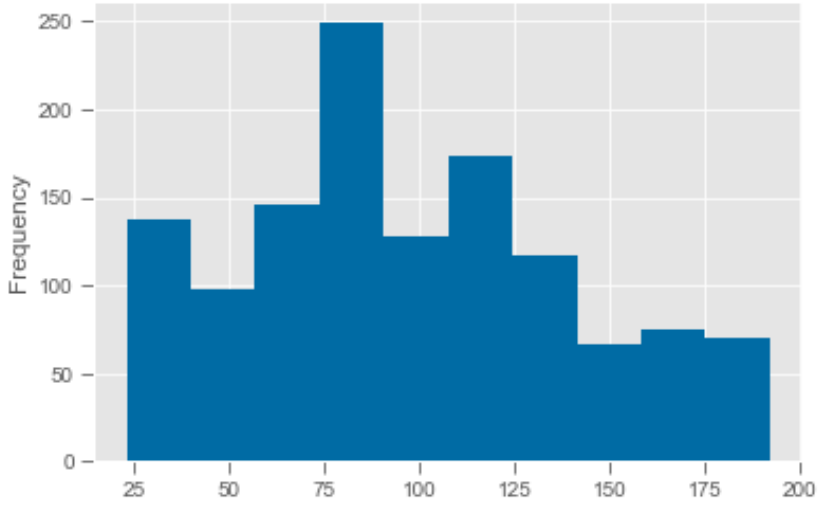


Visualization

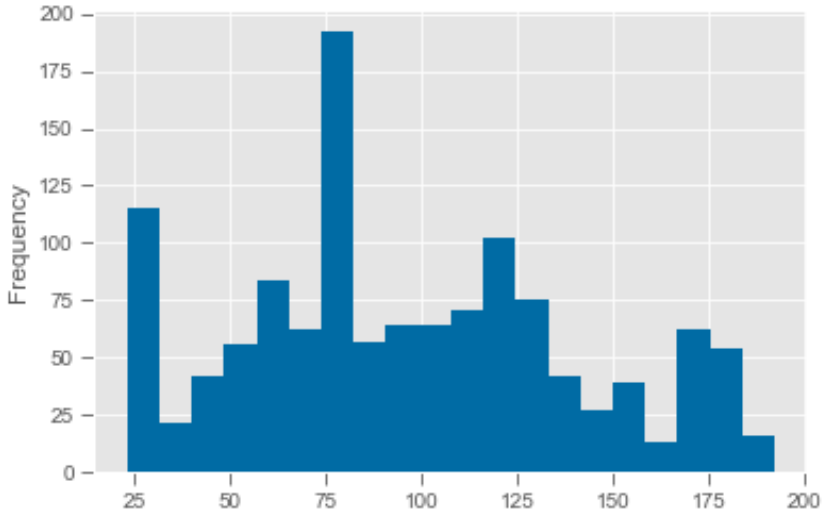
5. Histogram

- A histogram represents the frequency distribution of continuous variables.
- **kind = "hist"** represents a Histogram graph.
- **bins :** Number of histogram bins to be used. The bigger the number, the smaller the bar.

```
In [78]: FB["open"].plot(kind="hist", bins = 10)
Out[78]: <matplotlib.axes._subplots.AxesSubplot at 0x246d0d4c898
```




```
In [79]: FB["open"].plot(kind="hist", bins = 20)
Out[79]: <matplotlib.axes._subplots.AxesSubplot at 0x246d0d85be
```



Visualization

6. Scatter plot

- For this example, we will be using iris datasets. Iris datasets are well-known data to test with Machine Learning, Analysis, or Deep Learning.
- It is often used for testing out machine learning algorithms and visualizations.

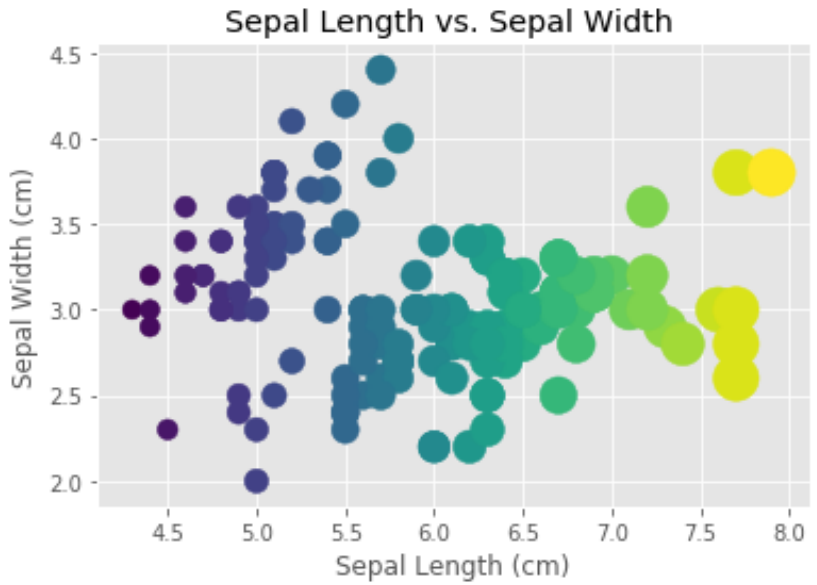
```
from sklearn import datasets

iris_data = datasets.load_iris()
iris_data.data
iris_data.target
iris_data.target_names

iris_df = pd.DataFrame(data=iris_data.data
                       , columns=iris_data.feature_names
                      ).join(
    pd.DataFrame(data=iris_data.target, columns=['Species']))

iris_df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	Species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

Visualization

7. Bubble plot

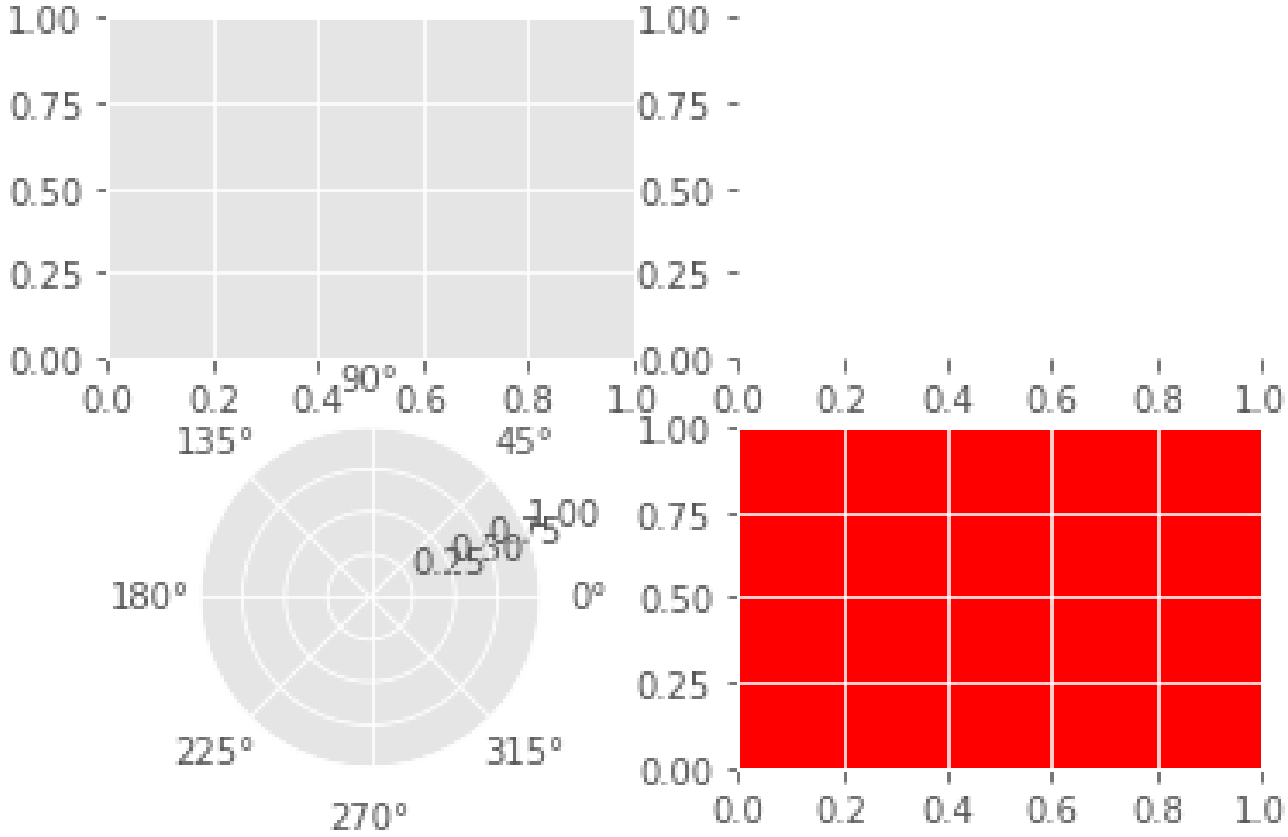
```
plt.scatter(x =iris_df["sepal length (cm)"] , y = iris_df["sepal width (cm)"] )
#add title
plt.title('Sepal Length vs. Sepal Width ')
#Name x label
plt.xlabel('Sepal Length (cm)')
#Name y label
plt.ylabel('Sepal Width (cm)')
plt.show()
```



Visualization

8.Subplots

- The **subplot()** method accepts parameters as shown:
  subplot(nrows, ncols, index)
- For example, **subplot(2,2,1)** stands for 2 rows, 2 columns and the graph at position 1.



Visualization

9. Saving the graph into an image file

- We can export graphs into any file type listed below:

Format	Format name
svgz	Scalable Vector Graphics
ps	Postscript
emf	Enhanced Metafile
rgba	Raw RGBA bitmap
raw	Raw RGBA bitmap
pdf	Portable Document Format
svg	Scalable Vector Graphics
eps	Encapsulated Postscript
png	Portable Network Graphics

- The code is very simple. The format parameter can be obtained from the table.

```
plt.savefig('Figure1.png')
```

```
<Figure size 432x288 with 0 Axes>
```

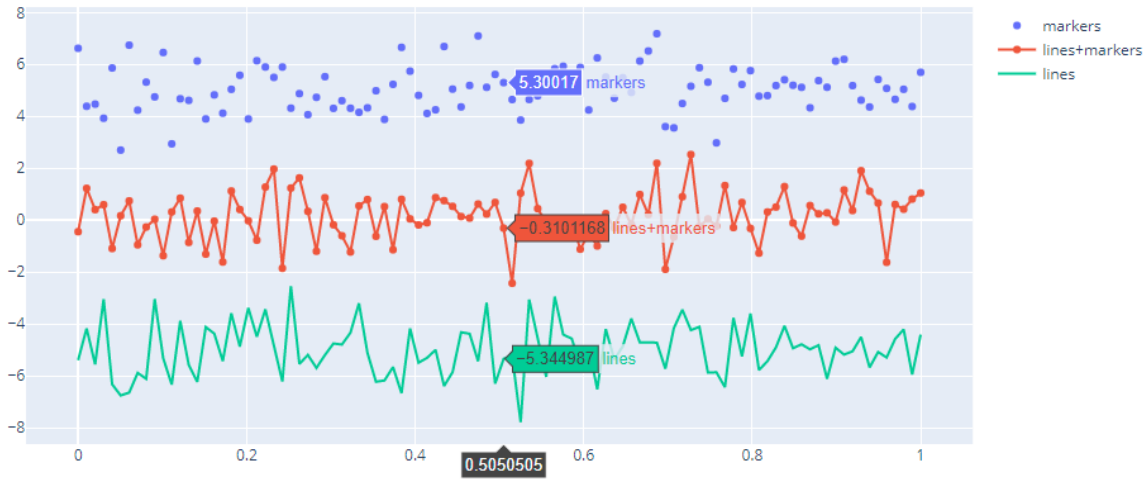
```
plt.savefig("Figure 2", format="png")
```

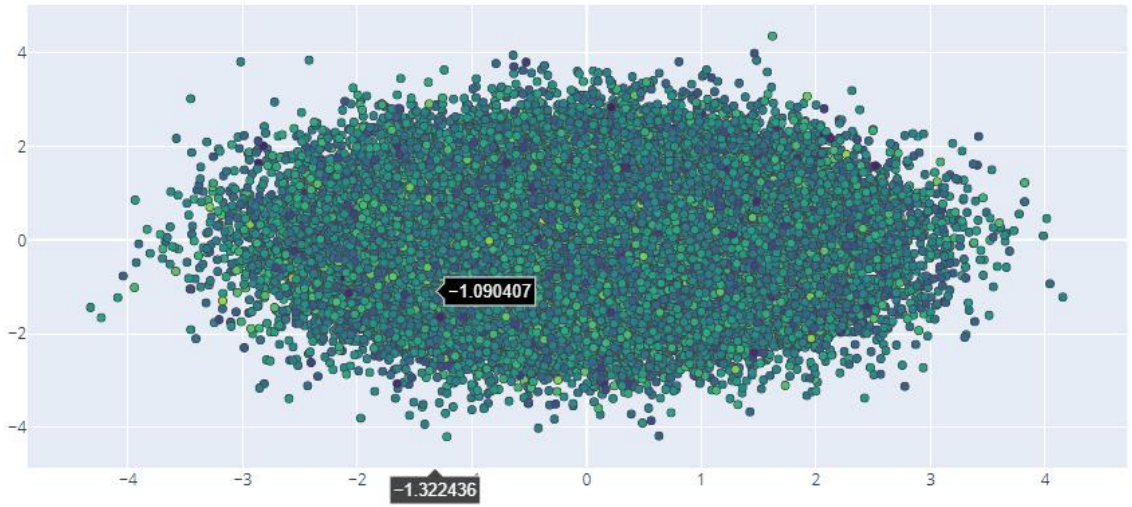
```
<Figure size 432x288 with 0 Axes>
```

Visualization

10. Visualizing the data using **Plotly**

- Plotly's Python graphing library makes **interactive, publication-quality graphs.** Simple graphs that can be created are line plots, scatter plots, area charts, and bar charts.
- Plotly visualizations can be exported using cloud and Dash Framework. Dash is a framework to build web applications. Dash is built on top of Flask (Python Framework) and can be used to create a simple dashboard for visualizations.

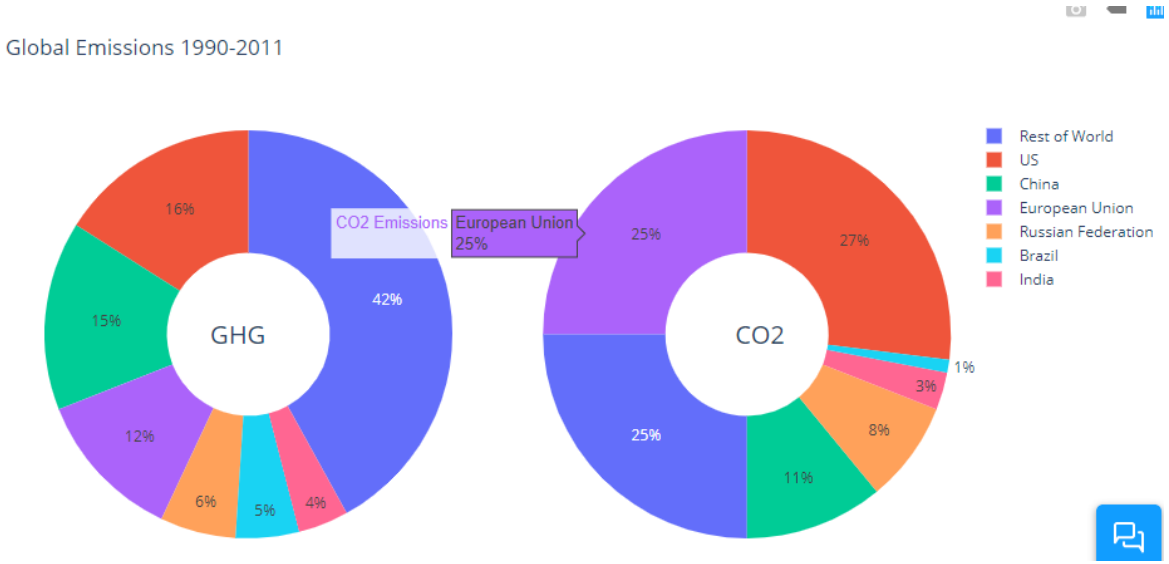




Visualization

10. Visualizing the data using Plotly : Example of Plotly plots

OHLC Graph



Pie Charts

Visualization

10. Visualizing the data using Plotly

- **Visualize the most expensive place for McDonalds.**
    1. The data is from 2010 to 2016. Hence, we need to get the lowest price for each Country.
    2. After that, we group the value by Country.
    3. Then, we create variable x and y to visualize the graph.
    4. Finally, we use Plotly library and Graph Objects to visualize the graph into Bar Chart.

Visualization

10. Visualizing the data using Plotly

- To plot a graph, we need the x and y variables. We use the **groupby()** method, to return the country as the index and mean price as the values. Hence, we can use **.index**, and **.values** attributes to get the x and y variables.

```
In [30]: temp = df.groupby("Country").mean()["Price in US Dollars"].sort_values(ascending=False)
         x = temp.head(10).index
         y = temp.head(10).values

         temp.head(10)
```

```
Out[30]: Country
         Norway          7.037500
         Switzerland     6.877500
         Sweden          5.955833
         Venezuela       5.549000
         Denmark         5.132500
         Brazil          5.110000
         Finland         4.978000
         France          4.767000
         Belgium         4.753000
         Italy           4.740000
         Name: Price in US Dollars, dtype: float64
```

Visualization

10. Visualizing the data using Plotly

- Next, create a **graph object** in which the x and y values will be passed to the object.
- Plotly has many parameters which can make the graph more beautiful and easier to read. The main components of the graph is the title, x label, and y label.

```
temp = df.groupby("Country").mean()["Price in US Dollars"].sort_values(ascending=False)
x = temp.head(10).index
y = temp.head(10).values

import plotly.graph_objects as go

fig = go.Figure(data=[go.Bar(x=x, y=y)])

layout = go.Layout(title="Top Expensive Big Mac by Counry",
                   xaxis = dict(title= "Coutry"),
                   yaxis = dict(title="Price of Big Mac"),


                  )

fig.update_layout(layout)
fig.show()
```

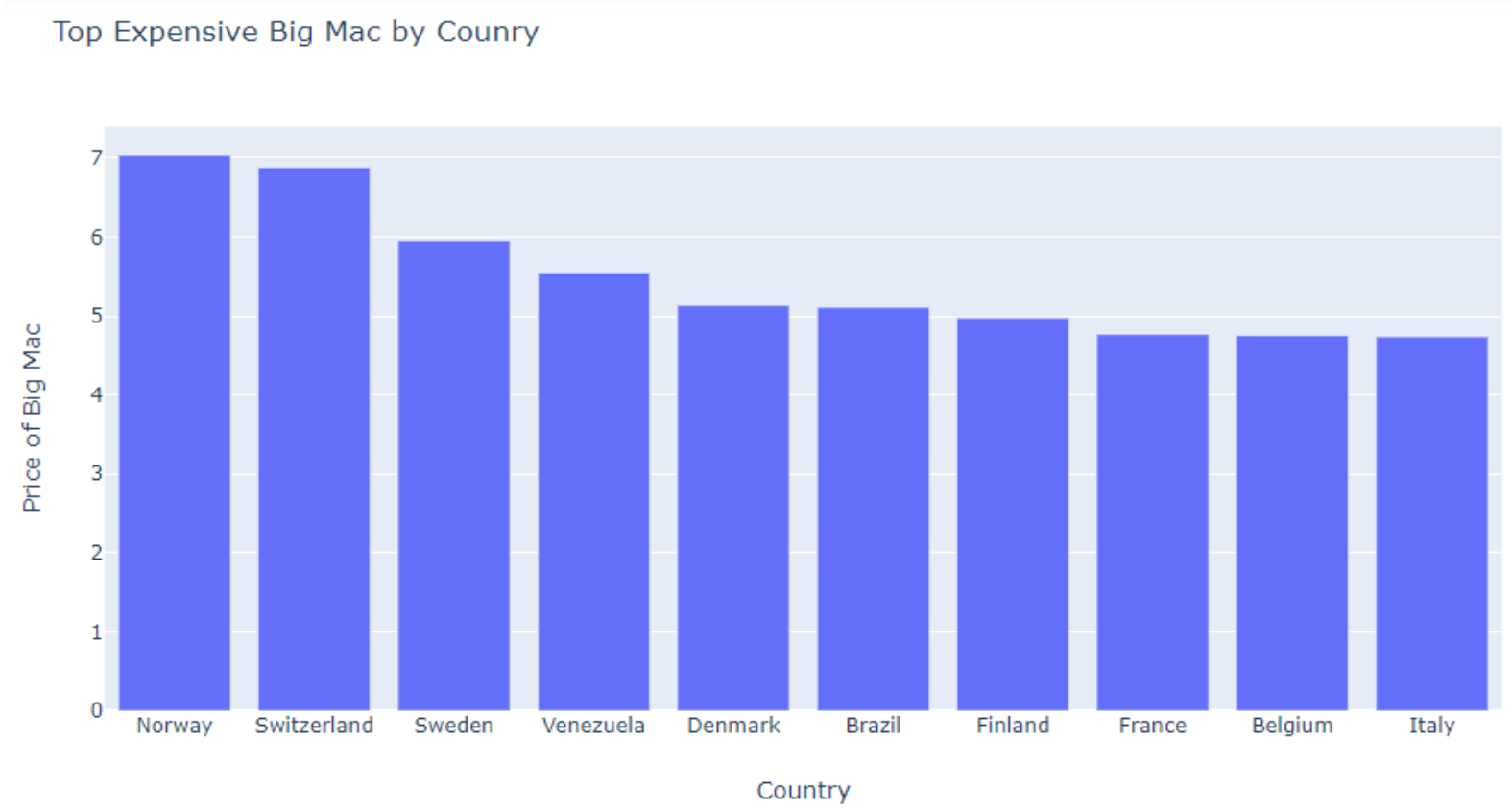
Graph values (x , y)

- Graph title
- x-axis label
- y-axis label

Visualization

10. Visualizing the data using Plotly

- Example of a simple graph.



Visualization

11. Scatter Plot Matrix using Plotly

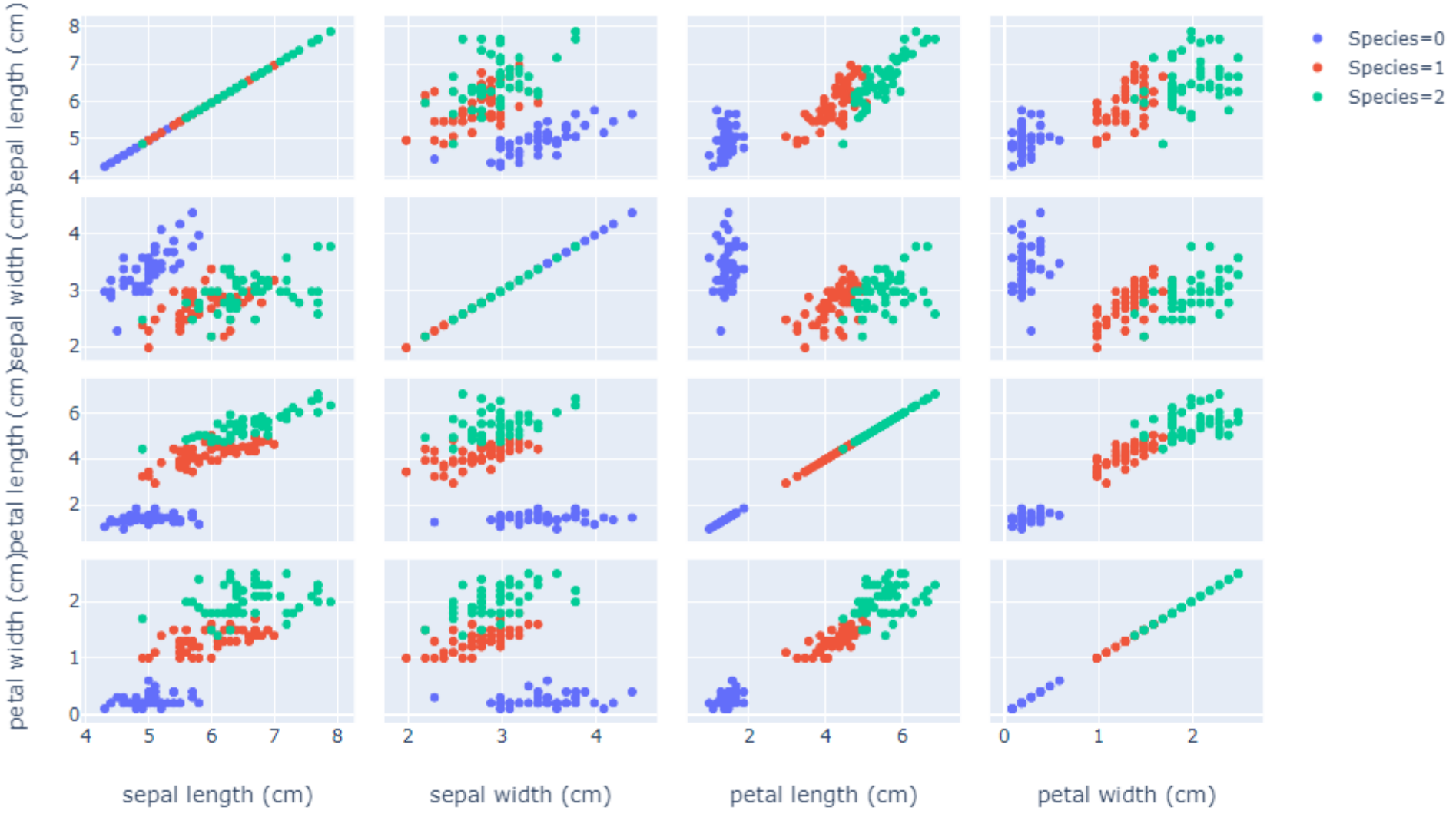
- In **seaborn**, this is known as a Pairplot. Although it has a different name, it functions the same. Using 3 to 4 lines of codes, we can visualize all the features. Scatter Matrix plot can only plot numerical columns. We use categorical columns to color the points.
- In the example, the "Species" column was used to color the values. The features can be understood much better.

```
In [37]: import plotly.express as px
         # iris = px.data.iris()
         iris_df["Species"] = iris_df["Species"].astype("category")

         fig = px.scatter_matrix(iris_df,
             dimensions=["sepal length (cm)", "sepal width (cm)" ,"petal length (cm)", "petal width (cm)"],
             color='Species'
                                 )
         fig.show()
```

The **color parameter** will color the points based on the **Species Columns**. In this way, we can understand compare each feature with group.

Visualization

11. Scatter Matrix Plot using Plotly



Visualization

12. Subplots using Plotly

- It is difficult to compare the graphs if they are not placed in the same figure. Hence, we can use subplots to plot many graphs at different positions.
- Subplots work like a 2D matrix. The graph is position by row and columns.

```
from plotly.subplots import make_subplots
import plotly.graph_objects as go

fig = make_subplots(rows=1, cols=2, subplot_titles=["Sepal Length (cm) X Sepal Width (cm)" ,
                                                    "Petal Length (cm) X Petal Width (cm)"])
```

- The **make_subplots()** method will create an object of subplots. We can specify the size of the subplots. We can also add titles for each subplot.

Visualization

12. Subplots using Plotly

- The **add_trace()** method is used to add the graph we want. The graph can be in **graph object** (we used go.Scatter).
- In the same method, we nee to specify the graph position (row=1, col = 1).
- The x and y labels should be added to each graph to make it more understandable. In the example, **update_xaxes()** and **update_yaxes()** was used to add the x and y labels respectively.

```
fig.add_trace(
    go.Scatter(x=iris_df["sepal length (cm)"],
               y=iris_df["sepal width (cm)"],
               mode='markers',
               marker = dict(color =iris_df['Species']),
               name = "Graph 1"),
    row=1, col=1
)

fig.add_trace(
    go.Scatter(x=iris_df["petal length (cm)"],
               y=iris_df["petal width (cm)"],
               mode='markers',
               marker = dict(color =iris_df['Species']),
               name = "Graph 2"),
    row=1, col=2
)


# Update yaxis properties
fig.update_yaxes(title_text="sepal length (cm)", row=1, col=1)
fig.update_yaxes(title_text="sepal width (cm)", row=1, col=2)

# Update xaxis properties
fig.update_xaxes(title_text="petal length (cm)", row=1, col=1)
fig.update_xaxes(title_text="petal width (cm)", row=1, col=2)

fig.update_layout(height=600, width=800, )
fig.show()
```

Visualization

12. Subplots using Plotly

- The generated graphs.