UNIT – III

Introduction to Anaconda -

Anaconda is a package manager, an environment manager, and Python distribution that contains a collection of many open source packages.

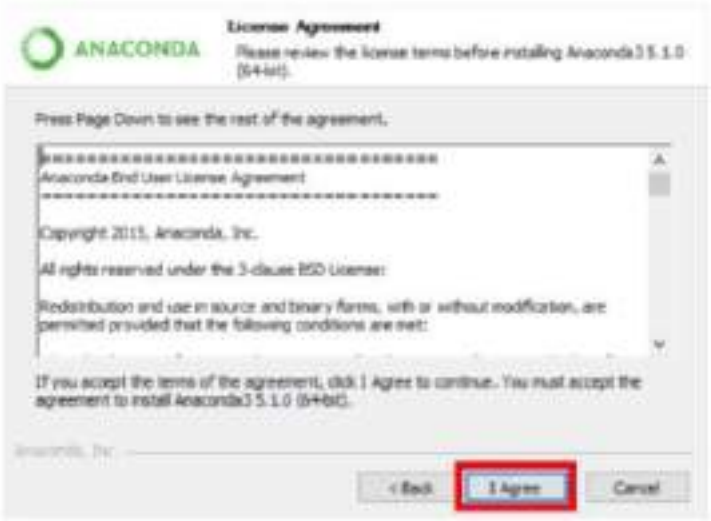
Anaconda Installation -

Go to the Anaconda Website and choose a Python 3.x graphical installer (A) or a Python 2.x graphical installer.

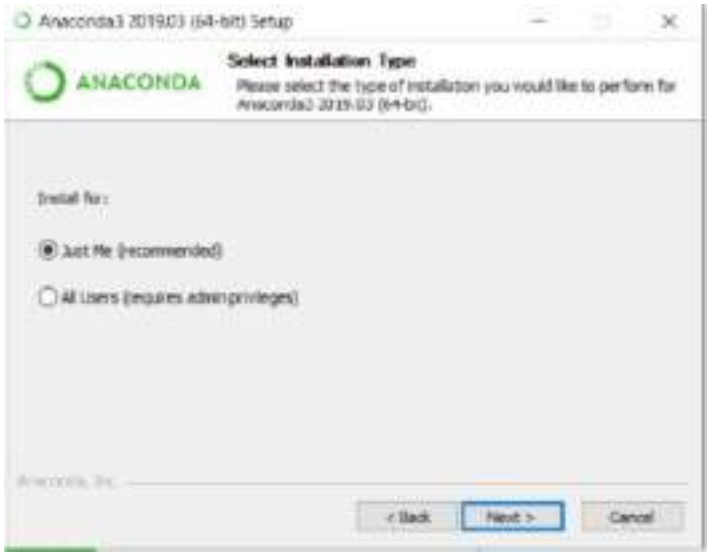
- Locate your download and double click it.

When the screen below appears, click on Next

- Read the license agreement and click on I Agree.



Click on Next.



Note your installation location and then click Next.

Choose whether to add Anaconda to your PATH environment variable. We recommend not adding Anaconda to the PATH environment variable, since this can interfere with other software. Instead, use Anaconda software by opening Anaconda Navigator or the Anaconda Prompt from the Start Menu.

After that click on next.

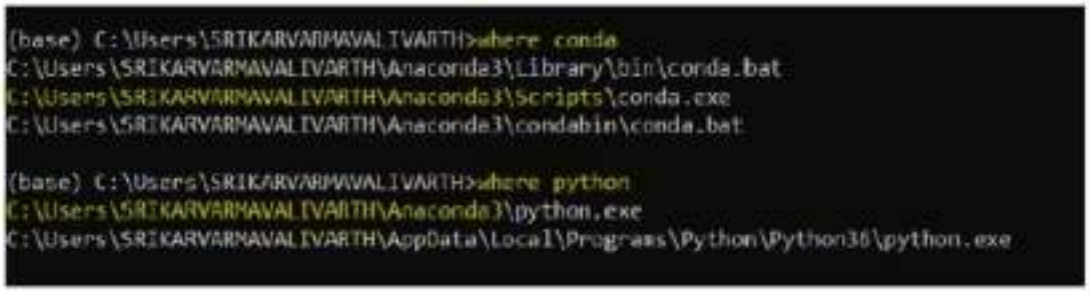
Click Finish.

We need to set anaconda path to system environmental variables.

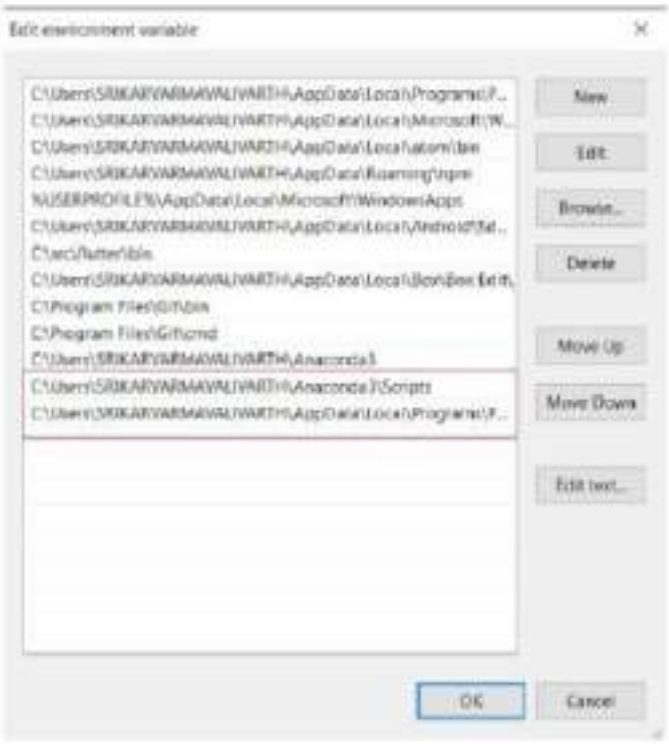
Open a Command Prompt. Check if you already have Anaconda added to your path.
Enter the commands below into your Command Prompt.

Conda –version
Python –version
This is checking if you already have Anaconda added to your path. If you get a command not recognized, then we need to set Anaconda path

If you don't know where your conda and/or python is, open an Anaconda Prompt and type in the following commands. This is telling you where conda and python are located on your computer.



Add conda and python to your PATH. You can do this by going to your System Environment Variables and adding the output of step 3 (enclosed in the red)



Open a **new Command Prompt.** Try typing conda --version and python --version into the **Command Prompt** to check to see if everything went well.

Conda installation is successful

Introduction to Jupyter Notebook

What is Jupyter

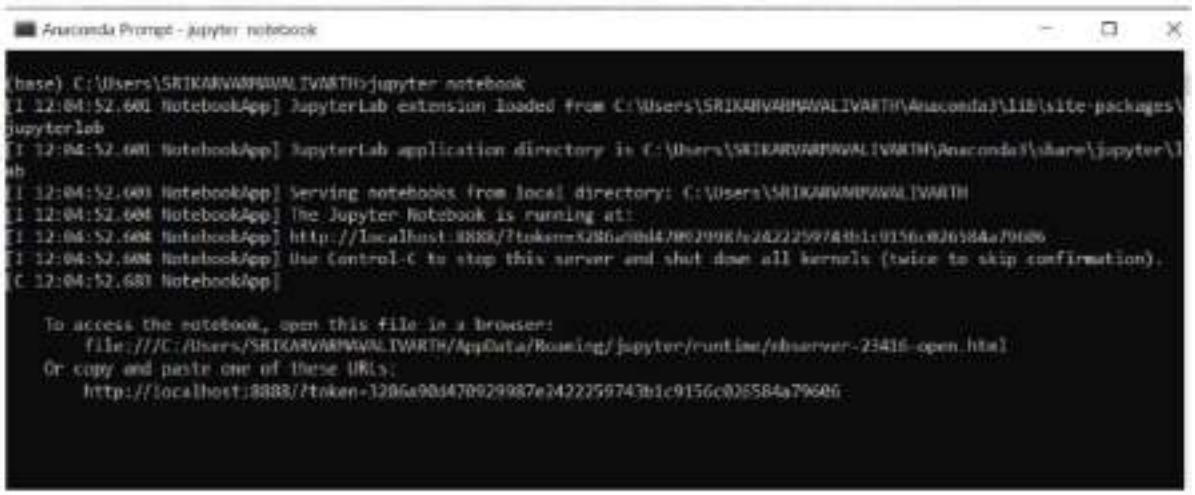
The Jupyter Notebook is an open source web application that you can use to create and share documents that contain live code, equations, visualizations, and text. Jupyter Notebook is maintained by the people at Project Jupyter.

Jupyter Notebooks are a spin-off project from the IPython project, which used to have an IPython Notebook project itself. The name, Jupyter, comes from the core supported programming languages that it supports: Julia, Python, and R. Jupyter ships with the IPython kernel, which allows you to write your programs in Python, but there are currently over 100 other kernels that you can also use.

How to access Jupyter Notebook

Installing Anaconda Distribution will also include Jupyter Notebook.

To access the Jupyter Notebook go to anaconda prompt and run below command



Or go to Command Prompt and first activate root before launching jupyter notebook

Then you'll see the application opening in the web browser on the following address: http://localhost:8888.

Python Scripting Basics
First Program in Python

```
>>> print ("Hello World !")
Hello World !
>>>
```

A statement or expression is an instruction the computer will run or execute. Perhaps the simplest program you can write is a print statement. When you run the print statement, Python will simply display the value in the parentheses. The value in the parentheses is called the argument.

If you are using a Jupyter notebook, you will see a small rectangle with the statement. This is called a cell. If you select this cell with your mouse, then click the run cell button. The statement will execute. The result will be displayed beneath the cell.

It’s customary to comment your code. This tells other people what your code does. You simply put a hash symbol proceeding your comment. When you run the code, Python will ignore the comment.

Data Types

A type is how Python represents different types of data. You can have different types in Python. They can be integers like 11, real numbers like 21.213. They can even be words.

11	int
21.213	float
"Hello Python 101"	str

type(11)	int
type(21.213)	float
type("Hello Python 101")	str

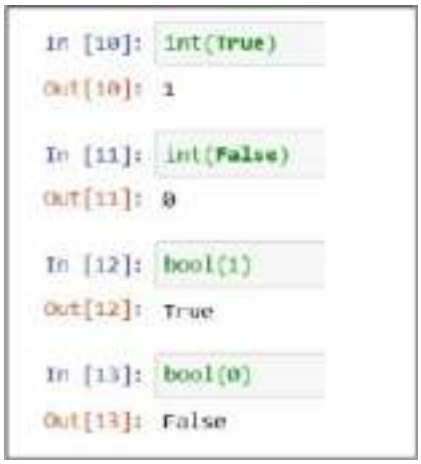
The following chart summarizes three data types for the last examples. The first coslumn indicates the expression. The second Column indicates the data type. We can see the actual data type in Python by using the type command. We can have int, which stands for an integer, and float that stands for float, essentially a real number. The type string is a sequence of characters.

Integers can be negative or positive. It should be noted that there is a finite range of integers, but it is quite large. Floats are real numbers; they include the integers but also numbers in between the integers. Consider the numbers between 0 and 1. We can select numbers in between them; these numbers are floats. Similarly, consider the numbers between 0.5 and 0.6. We can select numbers in-between them; these are floats as well.

Nothing really changes. If you cast a float to an integer, you must be careful. For example, if you cast the float 1.1 to 1, you will lose some information. If a string contains an integer value, you can convert it to int. If we convert a string that contains a non-integer value, we get an error. You can convert an int to a string or a float to a string.

Boolean is another important type in Python. A Boolean can take on two values. The first value is true, just remember we use an uppercase T. Boolean values can also be false, with an uppercase F. Using the type command on a Boolean value, we obtain the term bool, this is short for Boolean. If we cast a Boolean true to an integer or float, we will get a 1.

If we cast a Boolean false to an integer or float, we get a zero. If you cast a 1 to a Boolean, you get a true. Similarly, if you cast a 0 to a Boolean, you get a false.

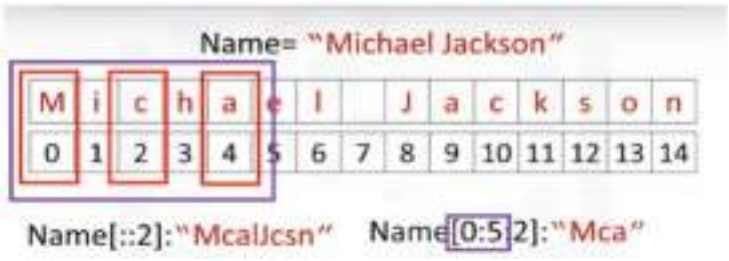


String Operations In Python

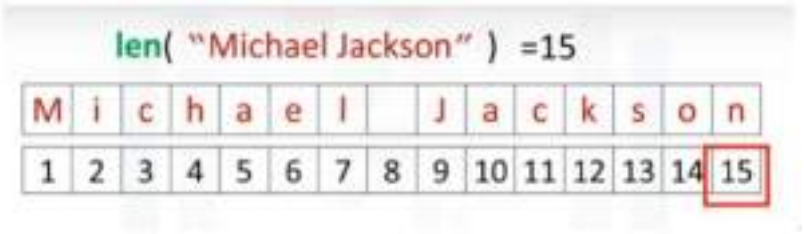
In Python, a string is a sequence of characters. A string is contained within two quotes: You could also use single quotes. A string can be spaces, or digits. A string can also be special characters. We can bind or assign a string to another variable. It is helpful to think of a string as an ordered sequence. Each element in the sequence can be accessed using an index represented by the array of numbers. The first index can be accessed as

follows. We can access index 6. Moreover, we can access the 13th index. We can also use negative indexing with strings. The last element is given by the index -1. The first element can be obtained by index -15, and so on.

We can bind a string to another variable. It is helpful to think of string as a list or tuple. We can treat the string as a sequence and perform sequence operations. We can also input a string value as follows. The 2 indicates we select every second variable. We can also incorporate slicing.



In this case. we return every second value up to index four. We can use the "Len" command to obtain the length of the string. As there are 15 elements, the result is 15.



We can concatenate or combine strings. We use the addition symbols. The result is a new string that is a combination of both.

We can replicate values of a string. We simply multiply the string by the number of times we would like to replicate it, in this case, three. The result is a new string. The new string consists

of three copies of the original string. This means you cannot change the value of the string, but you can create a new string.

Python COLLECTION (or)Arrays

There are four collection data types in the Python programming language:
Tuple is a collection which is ordered and unchangeable. Allows duplicate members.

List is a collection which is ordered and changeable. Allows duplicate members.

Set is a collection which is unordered and unindexed. No duplicate members.

Dictionary is a collection which is unordered, changeable and indexed. No duplicate members.

Tuple:
tuples are expressed as comma-separated elements within parentheses.

```
In [14]: Example_Tuple=(2,4,6,2,24,2345)
In [15]: Example_Tuple_2=("python",32,78.23)
In [16]: type(Example_Tuple_2)
Out[16]: tuple
```

Example_Tuple=(2,4,6,2,24,2345)

Example_Tuple=(2,4,6,2,24,2345)

INDEX	List Elements	negative Index
Example_Tuple[0]	2	Example_Tuple[-6]
Example_Tuple[1]	4	Example_Tuple[-5]
Example_Tuple[2]	6	Example_Tuple[-4]
Example_Tuple[3]	2	Example_Tuple[-3]
Example_Tuple[4]	24	Example_Tuple[-2]
Example_Tuple[5]	2345	Example_Tuple[-1]

```
In [4]: Example_Tuple[0]
Out[4]: 2
In [5]: Example_Tuple[-1]
Out[5]: 2345
In [10]: Example_Tuple[2:5]
Out[10]: (6, 2, 24)
In [8]: Example_Tuple[2:-1]
Out[8]: (2, 24)
```

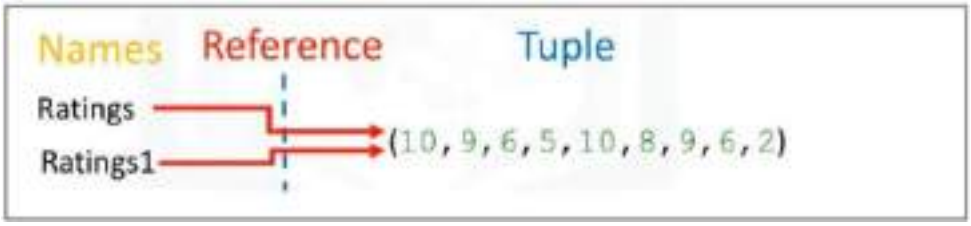
In Python, there are different types: strings, integer, float. They can all be contained in a tuple, but the type of the variable is tuple

Each element of a tuple can be accessed via an index. The element in the tuple can be accessed by the name of the tuple followed by a square bracket with the index number. Use the square brackets for slicing along with the index or indices to obtain value available at that index.

Tuples are immutable, which means we can't change them.

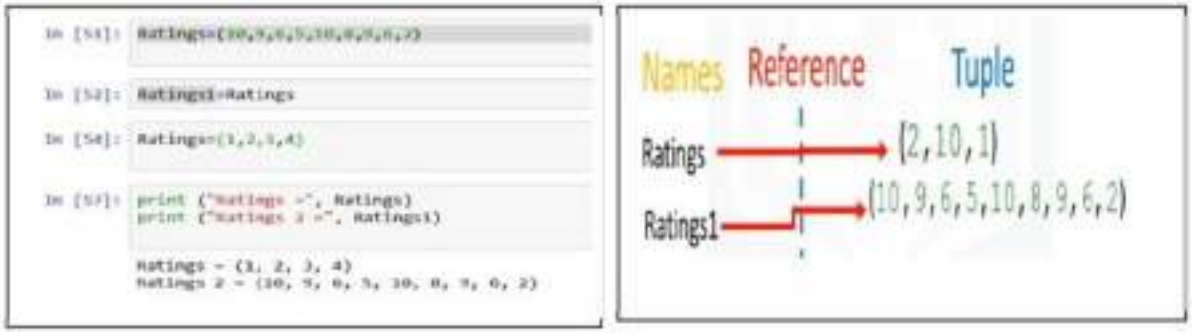
```
In [51]: Ratings=(10,9,6,5,10,8,9,6,2)

In [52]: Ratings1=Ratings

In [53]: print (Ratings1)
         (10, 9, 6, 5, 10, 8, 9, 6, 2)
```

To see why this is important, let's see what happens when we set the variable Ratings 1 to ratings. Each variable does not contain a tuple, but references the same immutable tuple object.



Let's say we want to change the element at index 2. Because tuples are immutable, we can't. Therefore, Ratings 1 will not be affected by a change in Rating because the tuple is immutable i.e., we can't change it.

We can assign a different tuple to the Ratings variable. The variable Ratings now references another tuple.



There are many built-in functions that take tuple as a parameter and perform some task. for example, we can find length of the tuple with len () function, minimum value with min () function... etc.

if we would like to sort a tuple, we use the function sorted. The input is the original tuple. The output is a new sorted tuple.

A tuple can contain other tuples as well as other complex data types; this is called nesting.

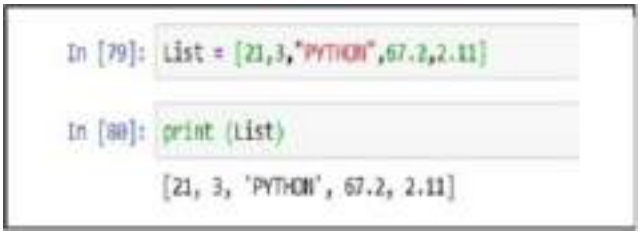
For Example: NestedTuple = (5,2, ("A","B"),(1,2),(8896,("x","y","z")))
We can access these elements using the standard indexing methods.

For example, we could access the second element. We can apply this indexing directly to the tuple variable NT. It is helpful to visualize this as a tree. We can visualize this nesting as a tree. The tuple has the following indexes. If we consider indexes with other tuples, we see the tuple at index 2 contains a tuple with two elements. We can access those two indexes. The same convention applies to index 3. We can access the elements in those tuples as well. We can

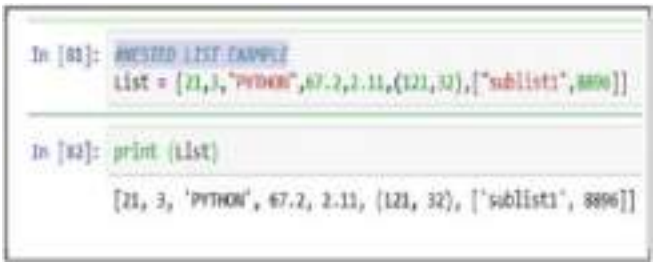
continue the process. We can even access deeper levels of the tree by adding another square bracket like NestedTuple

List:

A list is a collection which is ordered and changeable. A list is represented with square brackets. In many respects' lists are like tuples, one key difference is they are mutable. Lists can contain strings, floats, integers We can nest other lists.



We can also nest tuples and other data structures; the same indexing conventions apply for nesting Like tuples, each element of a list can be accessed via an index.



List = [21,3,"PYTHON",67.2,2.11,(121,32),["sublist1",8896]]

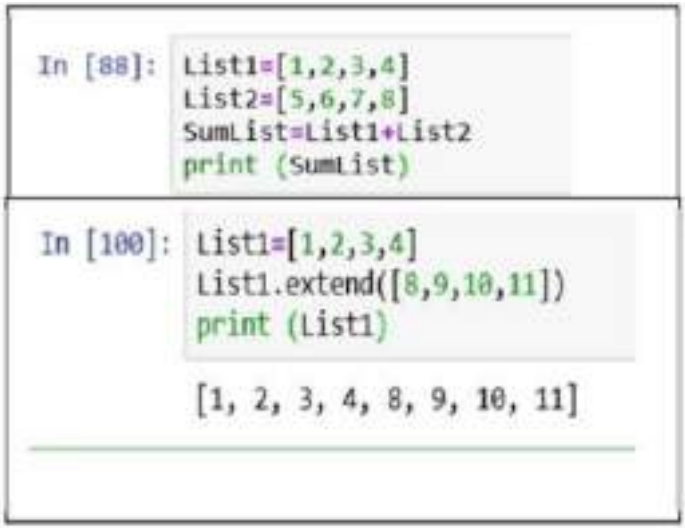
INDEX	List ELEMENTS	Negetive INDEX
List[0]	21	List[-7]
List[1]	3	List[-6]
List[2]	PYTHON	List[-5]
List[3]	67.2	List[-4]
List[4]	2.11	List[-3]
List[5]	(121,32)	List[-2]
List[6]	["subList1",8896]	List[-1]

The following table represents the relationship between the index and the elements in the list. The first element can be accessed by the name of the list followed by a square bracket with the index number, in this case zero. We can access the second element as follows. We can also access the last element. In Python, we can use a negative index.

The index conventions for lists and tuples are identical for accessing and slicing the elements.

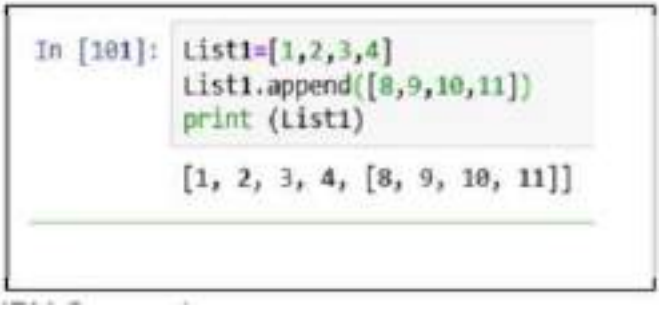
We can concatenate or combine lists by adding them. Lists are mutable; therefore, we can

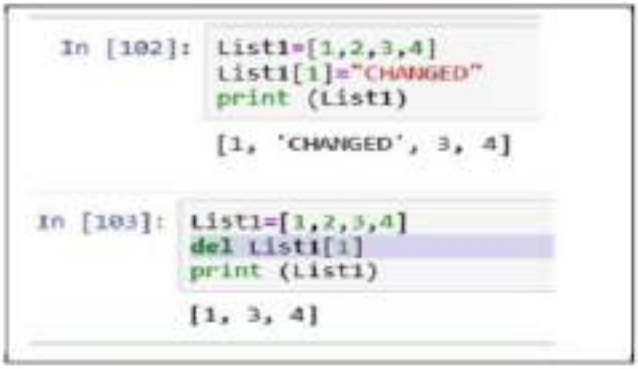
change them. For example, we apply the method Extends by adding a "dot" followed by the name of the method, then parenthesis.



The argument inside the parenthesis is a new list that we are going to concatenate to the original list. In this case, instead of creating a new list, the original list List1 is modified by adding four new elements.

Another similar method is append. If we apply append instead of extended, we add one element to the list. If we look at the index, there is only one more element. Index 4 contains the list we appended.
Every time we apply a method, the lists changes.





As lists are mutable, we can change them. For example, we can change the Second element as

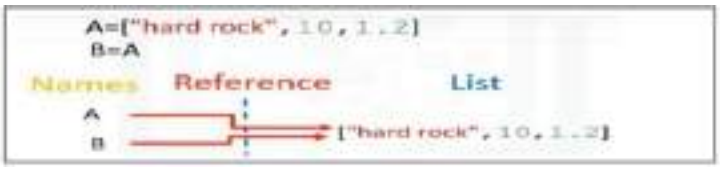
follows. The list now becomes [1," CHANGED",3,4]

We can delete an element of a list using the "del" command; we simply indicate the list item we would like to remove as an argument.

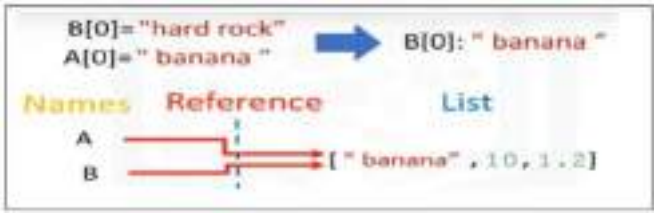
For example, if we would like to remove the Second element, then perform del List [1] command This operation removes the second element of the list then the result becomes [1,3,4]

LISTS: Aliasing

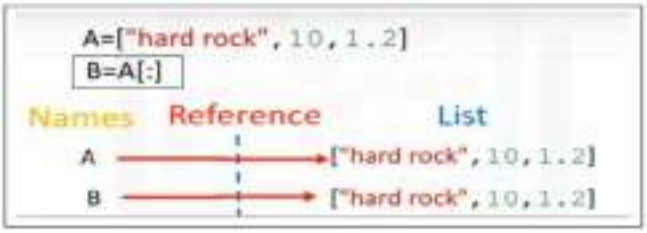
When we set one variable, B equal to A, both A and B are referencing the same list. Multiple names referring to the same object is known as aliasing.



If we change the first element in "A" to "banana" we get a side effect; the value of B will change as a consequence. "A" and "B" are referencing the same list, therefore if we change "A", list "B" also changes. If we check the first element of B after changing list "A" we get banana instead of hard rock



You can clone list "A" by using the following syntax. Variable "A" references one list. Variable "B" references a new copy or clone of the original list.



Now if you change "A", "B" will not change We can get more info on lists, tuples and many other objects in Python using the help command.

Simply pass in the list, tuple or any other Python object example: help(list),help(tuple)..etc.

Set:

Sets are a type of collection. Unlike lists and tuples, they are unordered. You cannot access items in a set by referring to an index, since sets are unordered the items has no index. To define a set, you use curly brackets You place the elements of a set within the curly brackets.

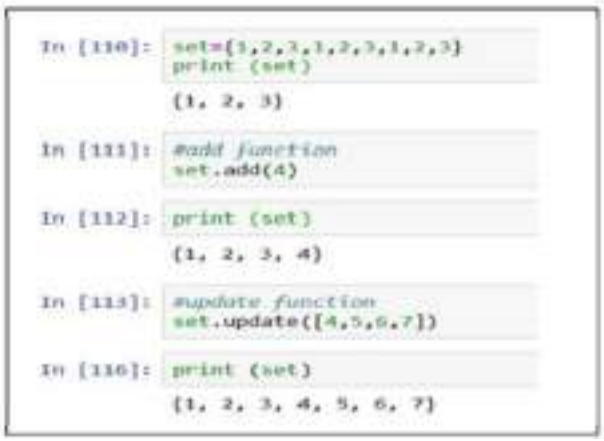
You notice there are duplicate items. When the actual set is created, duplicate items will not be present.

To add one item to a set, use the add () method.

To add more than one item to a set use the update () method with list of values.

To remove an item from the set we can use the pop () method. Remember sets are unordered so it will remove the first item in the set.

To remove an item from the set, use the remove method, we simply indicate the set item we would like to remove as an argument.



```
In [117]: set={1,2,3,1,2,3,1,2,3,4,5,6}
          print (set)

          {1, 2, 3, 4, 5, 6}

In [118]: #pop function
          set.pop()

Out[118]: 1

In [119]: print (set)

          {2, 3, 4, 5, 6}

In [120]: #remove function
          set.remove(5)

In [121]: print (set)

          {2, 3, 4, 6}
```

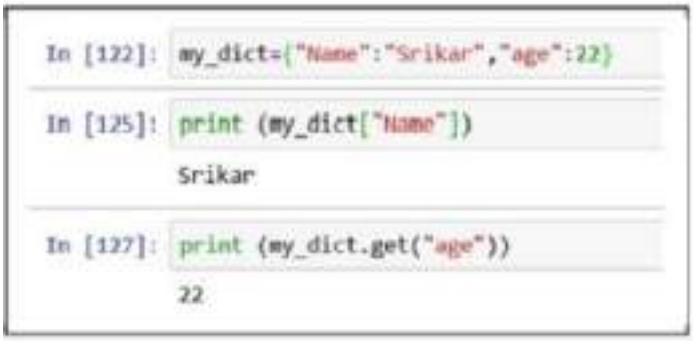
There are lots of useful mathematical operations we can do between sets. like union, intersection, difference, symmetric difference from two sets.

DICTIONARIES:

Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair. Dictionaries are optimized to retrieve values when the key is known. Creating a dictionary is as simple as placing items inside curly braces {} separated by comma. An item has a key and the corresponding value expressed as a pair, key: value. While values can be of any data type and can repeat, keys must be of immutable type (**string, number or tuple** with immutable elements) and must be unique.

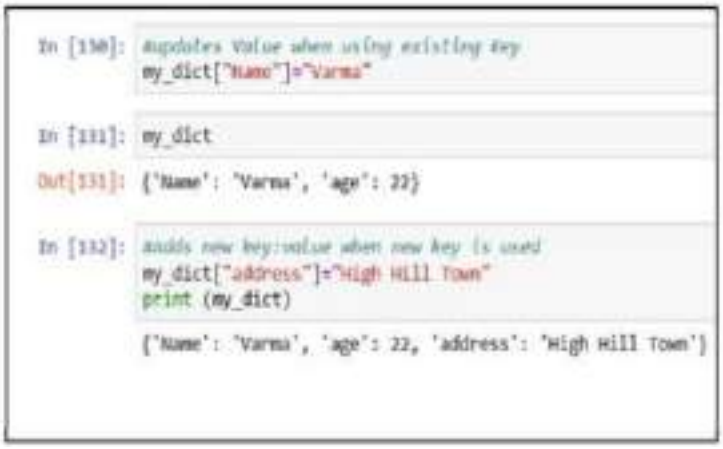
```
In [122]: my_dict={"Name":"Srikar","age":22}

In [123]: #my_dict variable is now refering to Dictionary object
          type(my_dict)

Out[123]: dict
```

We can access the elements from the dictionary using keys.



We can get the value using keys either inside square brackets or with get( ) method.

Dictionary is mutable. We can add new items or change the value of existing items using assignment operator. If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

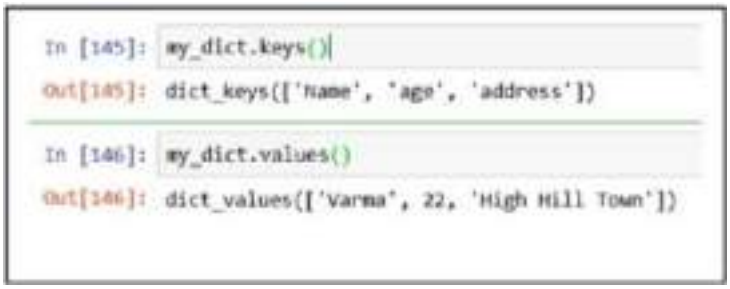


We can delete an entry as follows. This gets rid of the key "address" and its value from my_dict dictionary.

We can verify if an element is in the dictionary using the in command as follows.
Syntax: 'KEY_NAME' in DictionaryName
The command checks the keys. If they are in the dictionary, they return a true. If we

try the same command with a key that is not in the dictionary, we get a false. If we try with another key that is not in the dictionary, we get a false.

In order to see all the keys in a dictionary, we can use the method keys to get the keys. The output is a list like object with all keys. In the same way, we can obtain the values.



Conditional Statements

What is Control or Conditional Statements -

In programming languages, most of the time we have to control the flow of execution of your program, you want to execute some set of statements only if the given condition is satisfied, and a different set of statements when it's not satisfied. Which we also call it as control statements or decision-making statements.

Conditional statements are also known as decision-making statements. We use these statements when we want to execute a block of code when the given condition is true or false.

Usually Condition will be in a form of Expression with some relational operators. Refer some below operators mentioned in the chart

Meaning	Operator
Equal to	==
Greater than	>
Less than	<
Greater than or equal to	>=
Less than or equal to	<=
Not equal	!=
Negation	not

In Python we achieve the decision-making statements by using below statements -

If statements

If-else statements

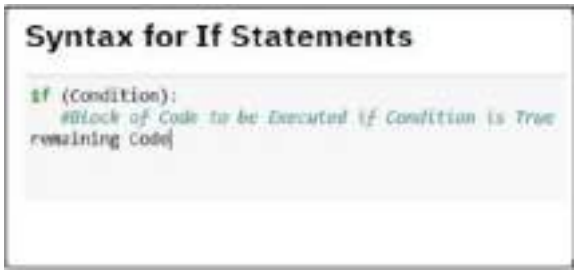
Elif statements

Nested if and if-else statements

If statements -

If statement is one of the most commonly used conditional statement in most of the programming languages. It decides whether certain statements need to be executed or not. If statement checks for a given condition, if the condition is true, then the set of code present inside the if block will be executed.

The If condition evaluates a Boolean expression and executes the block of code only when the Boolean expression becomes TRUE. Check the Syntax first the controller will come to an if condition and evaluate the condition if it is true, then the statements will be executed, otherwise the code present outside the block will be executed.



Let’s take an example to implement the if statement, in this example we have a variable name which stores the string “Srikar” and we also have names list with some names

Example

```
In [151]: name="Srikar"
          names=["Srikar","Harish","Vara"]
          if(name in names):
              print ("your name is present")
          print ("this statement will always be Executed")

          your name is present
          this statement will always be Executed
```

We can use if statement to check whether the name is present in the names list or not, if condition is true then it will also print block of statements inside the 'if' block. If condition is false, then it will skip the execution of the 'if' block statements.

If-else statements:

The statement itself tells that if a given condition is true then execute the statements present inside if block and if the condition is false then execute the else block.

Else block will execute only when the condition becomes false, this is the block where you will perform some actions when the condition is not true.

If-else statement evaluates the Boolean expression and executes the block of code present inside the if block if the condition becomes TRUE and executes a block of code present in the else block if the condition becomes FALSE.

Syntax for if-else statement

```
if (Condition):
    #Block of Code to be Executed if Condition is True
else:
    #Block of code to be Executed if condition is false
remaining Code
```

Let's take an example to implement the if-else statement, in this example the if block will get executed if the given condition is true or else it will execute the else block.

Example

```
In [152]: num=4
          if(num>5):
              print("number is greater than 5")
          else:
              print("number is less than 5")

          number is less than 5
```

elif statements:

In python, we have one more conditional statement called elif statements. Elif statement is used to check multiple conditions only if the given if condition false. It's like an if-else statement and the only difference is that in else we will not check the condition but in elif we will do check the condition.

Syntax for elif statement

```
if (condition):
        #Set of statement to execute if condition is true
elif (condition):
        #Set of statements to be executed when if condition is false and elif condition is true
else:
    #Set of statement to be executed when both if and elif conditions are false
```

Elif statements are similar to if-else statements but elif statements evaluate multiple conditions.

Example

```
In [153]: num=4
          if (num==0):
              print("number is zero")
          elif (num > 5):
              print("number is greater than 5")
          else:
              print("number is less than 5")

          number is less than 5
```

Let’s take an example to implement the elif statement, in this example the if block will get executed if the given if-condition is true, or elif block will get executed if the elif-condition is true, or it will execute the else block if both if and elif conditions are false.

Nested if-else statements

Nested if-else statements mean that an if statement or if-else statement is present inside another if or if-else block. Python provides this feature as well, this in turn will help us to check multiple conditions in a given program. An if statement present inside another if statement which is present inside another if statements and so on.

Syntax for nested if-else

```
if(condition):
        #statements to execute if condition is true
        if(condition):
                #statements to execute if condition is true
        else:
                #Statements to execute if condition is false
else:
        #statements to execute if condition is false
```

Example for nested if-else

```
In [164]: num=-5
          if(num!=0):
              print("number not equal to 0")
              if(num>0):
                  print("number is positive")
              else:
                  print("number is negitive")
          else:
              print("number is equal to 0")

          number not equal to 0
          number is negitive
```

Numpy and Pandas > Numpy overview - Creating and Accessing Numpy Arrays > Numpy overview - Creating and Accessing Numpy Arrays

What is numpy ?

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms,

basic linear algebra, basic statistical operations, random simulation and much more.

Difference between numpy arrays and lists

There are several important differences between NumPy arrays and the standard Python sequences, NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically).

The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.

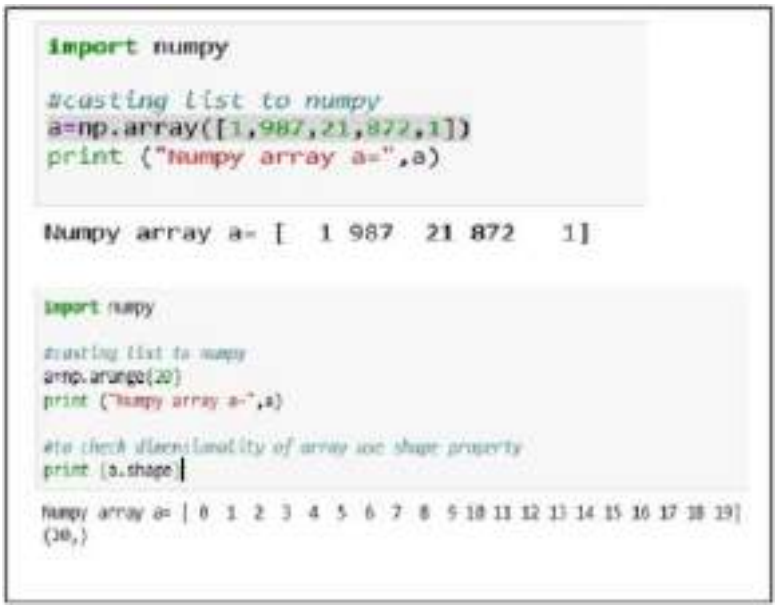
NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.

CREATING NUMPY 1D ARRAY

A "numpy" array or "ndarray" is similar to a list. It's usually fixed in size and each element is of the same type, we can cast the list to numpy array by first importing the numpy. Or We can also quickly create the numpy array with arange function which creates an array within the range specified.

To verify the dimensionality of this array, use the shape property.

In example Since there is no value after the comma (20,) this is a one-dimensional array.



Accessing NUMPY 1D ARRAY

```
In [177]: a=np.array([1,987,21,872,1])
          print ("Numpy array a=",a)

          #accessing first element of numpy array
          print ("a[0] =",a[0])

          #change the 2nd value of numpy array to 1000
          a[1]=1000
          print ("After changing 2nd value to 100 = ",a)

          #slicing the first 3 values from numpy array
          print ("After slicing  =",a[0:3])

          Numpy array a= [  1 987  21 872   1]
          a[0] = 1
          After changing 2nd value to 100 =  [   1 1000   21  872    1]
          After slicing  = [   1 1000   21]
```

Accessing and slicing operations for 1D array is same as list. Index values starts from 0 to length of the list.

Creating numpy 2D ARRAY

If you only use the arrange function, it will output a one-dimensional array. To make it a two-dimensional array, chain its output with the reshape function.

In this example first, it will create the 15 integers and then it will convert to two dimensional array with 3 rows and 5 columns.

```
a=np.arange(15).reshape(3,5)
print (a)

print ("SHAPE OF THE NUMPY ARRAY IS =", a.shape)

[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
SHAPE OF THE NUMPY ARRAY IS = (3, 5)
```

Accessing NUMPY 2D ARRAY

To access an element in a two-dimensional array, you need to specify an index for both the row and the column.

```
: a=np.arange(15).reshape(3,5)
  print (a)

  print ("Element in 1nd Row and 1 coloumn =", a[0,0])
  print ("Element in 2nd Row and 5 coloumn =", a[1,4])
  print ("Element in 3nd Row and 5 coloumn =", a[2,4])
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
Element in 1nd Row and 1 coloumn = 0
Element in 2nd Row and 5 coloumn = 9
Element in 3nd Row and 5 coloumn = 14
```

Introduction to Pandas

What are pandas ?

Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures. Pandas is the backbone for most of the data projects.

Through pandas, you get acquainted with your data by cleaning, transforming, and analyzing it. Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.

We can import the library or a dependency like pandas using the **"import pandas"** command. We now have access to many pre-built classes and functions.

In order to be able to work with the data in Python, we'll need to read the data(csv, excel ,dictionary,..) file into a **Pandas DataFrame**. A DataFrame is a way to represent and work with tabular data. Tabular data has rows and columns, just like our csv file. In order to read in the data, we'll need to use the **pandas.read_csv** function. This function will take in a csv file and return a DataFrame.

What is csv ?

csv stands for comma-separated values, csv file is a delimited text file that uses a comma to separate values. A CSV file stores tabular data in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas.

How to read the csv file using pandas

Once the pandas library is imported, This assumes the library is installed. Then we can load a csv file using the pandas built-in function "read csv." A csv is a typical file type used to store data.

We simply type the word pandas, then a dot and the name of the function with all the inputs. Typing pandas all the time may get tedious.

We can use the "as" statement to shorten the name of the library; in this case we use the standard abbreviation pd. Now we type pd and a dot followed by the name of the function we would like to use, in this case, read_csv.

we need to give the path of the csv file as argument to the read_csv function, to read the path string correctly we need to use ‘r’ as a prefix to the command. The result is stored in the variable df. this is short for “dataframe." Now that we have the data in a dataframe, we can work with it. We can use the method head to see the entire data frame or we can pass the number of rows to be checked as an argument to the head method like df.head(5) for 5 rows.

How to Write the csv file using pandas

If you want to write the dataframe to csv we can simple use the “to_csv” function

Syntax:
df.to_csv(EXPORT FILE PATH)

example:
If you see the above data frame example we see the two indexes one is loaded from the csv file and also there is unnamed index which is by default generated by pandas while loading the csv. This problem can be avoided by making sure that the writing of CSV files doesn’t write indexes, because DataFrame will generate it anyway. We can do the same by specifying index = False parameter in to_csv(...) function.

df.to_csv(‘EXPORT FILE PATH’, index=False)

Descriptive statistics using pandas

There are many collective methods to compute descriptive statistics and other related operations on pandas DataFrame.

Steps TO FOLLOW FOR Descriptive Statistics

These are the three steps we should perform to do statistical analysis on pandas dataframe.

collect the data

create the data frame

get the descriptive statistics for pandas dataframe

Collect the data:

To do any statistical analysis, first collection of data is the important task

You can store the collected data in csv, excel, or in dictionary format. For Demo we store the home data in one csv file.

Create the data frame:

we need to create the data frame based on the data collected.
Give the homes csv file path location.

```
In [267]: import pandas as pd
          df=pd.read_csv(r"C:\Users\SRIKARVARMAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\homes.csv")
          df.head(5)
```

Out[267]:

	Sell	"Rooms"	"Beds"	"Acres"	"Taxes"
0	142	10.0	5.0	0.28	3167.0
1	175	8.0	4.0	0.43	4033.0
2	129	6.0	3.0	0.33	1471.0
3	138	7.0	3.0	0.46	3204.0
4	232	8.0	4.0	2.05	3613.0

Once you run the above code you will get this DataFrame.

Get the Descriptive Statistics for Pandas DataFrame

Once you have your Data Frame ready, you'll be able to get the Descriptive Statistics. We can calculate the following statistics using the pandas package:

Mean

Total sum

Maximum

Minimum

Count

Median

Standard deviation

Variance

With describe function you will get complete descriptive stats

The syntax is: df.describe()

```
In [320]: df.describe()
```

Out[320]:

	Sell	Rooms	Beds	Acres	Taxes
count	9.000000	9.000000	9.000000	9.000000	9.000000
mean	175.444444	8.000000	4.000000	1.207778	3611.888889
std	50.356507	1.322876	0.707107	1.284132	1247.810327
min	129.000000	6.000000	3.000000	0.280000	1471.000000
25%	138.000000	7.000000	4.000000	0.430000	3131.000000
50%	150.000000	8.000000	4.000000	0.530000	3204.000000
75%	207.000000	8.000000	4.000000	2.050000	4033.000000
max	271.000000	10.000000	5.000000	4.000000	5702.000000

We can also get the descriptive statistics for the 'particular field:

```
In [321]: df["Rooms"].describe()
```

```
Out[321]: count     9.000000
          mean      8.000000
          std       1.322876
          min       6.000000
          25%       7.000000
          50%       8.000000
          75%       8.000000
          max      10.000000
          Name: Rooms, dtype: float64
```

You can further breakdown the descriptive statistics into the following measures:

	Sell	Rooms	Beds	Acres	Taxes
0	142	10	5	0.28	3167
1	175	8	4	0.43	4033
2	129	6	3	0.33	1471
3	138	7	3	0.46	3204
4	232	8	4	2.05	3613
5	135	7	4	0.57	3028
6	150	8	4	4.00	3131
7	207	8	4	2.22	5158
8	271	10	5	0.53	5702

```
In [330]: #To get minimum value for "Sell" Coloumn
          df['Sell'].min()

Out[330]: 129

In [332]: #To get maximum value for "Rooms" Coloumn
          df['Rooms'].max()

Out[332]: 10

In [333]: #To calculate the mean for "Acres" Coloumn
          df['Acres'].mean()

Out[333]: 1.2077777777777778
```

Pandas working with text data and datetime columns

While working with data, it is not an unusual thing to encounter time series data. Working with datetime columns can be quite challenge task. Luckily, pandas are great at handling time series data. Pandas provide a different set of tools using which we can perform all the necessary tasks on date-time data.

Let's see how we can convert a dataframe column of strings (in dd/mm/yyyy format) to datetime format. We cannot perform any time series-based operation on the dates if they are not in the right format. To be able to work with it, we are required to convert the dates into the datetime format.

Convert Pandas dataframe column type from string to datetime format

For any operation we need to first create the data frame based on the data collected, we can load the data either from csv file, or excel file or from any source. Let us use csv file for our

demo.

Follow the below lines of code to load the data and convert that to data Frame.

Once the data frame is ready use df.info( ) to get complete information of dataframe.

```
In [343]: import pandas as pd
          df=pd.read_csv(r"C:\Users\SRIKARVARMAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\DateTime.csv")
          df.head(5)
```

Out[343]:

	DateTime	QueueName	Q_Used	TotalMemoryGB	FreeMemoryGB	Processors	FreeMemoryPercentage
0	5/23/2018 15:42	6	0	1.38	0.31	2	22.57
1	5/23/2018 15:43	6	0	1.38	0.31	2	22.52
2	5/23/2018 15:44	6	0	1.38	0.30	2	21.51
3	5/23/2018 15:45	6	0	1.38	0.30	2	21.46
4	5/23/2018 15:46	6	0	1.38	0.29	2	21.25

```
In [344]: df.info(
          )
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 46524 entries, 0 to 46523
Data columns (total 7 columns):
DateTime                3778 non-null object
QueueName               46524 non-null int64
Q_Used                  46524 non-null int64
TotalMemoryGB           46524 non-null float64
FreeMemoryGB            46524 non-null float64
Processors              46524 non-null int64
FreeMemoryPercentage    46524 non-null float64
dtypes: float64(3), int64(3), object(1)
memory usage: 2.54 MB
```

As we can see in the output, the data type of the 'DateTime' column is object i.e. string. Now we will convert it to datetime format using **pd.to_datetime()** function.

```
In [345]: df['DateTime']= pd.to_datetime(df['DateTime'])

In [346]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 46524 entries, 0 to 46523
Data columns (total 7 columns):
DateTime                3778 non-null datetime64[ns]
QueueName               46524 non-null int64
Q_Used                  46524 non-null int64
TotalMemoryGB           46524 non-null float64
FreeMemoryGB            46524 non-null float64
Processors              46524 non-null int64
FreeMemoryPercentage    46524 non-null float64
dtypes: datetime64[ns](1), float64(3), int64(3)
memory usage: 2.5 MB
```

After applying the pd.to_datetime () function to DateTime column, we can see in the output, the format of the 'DateTime' column has been changed to the datetime format.

HOW TO CHANGE THE INDEX of the dataframe

Most of the operations related to dateTime requires the DateTime column as the primary index, or else it will throw an error.

We can change the index with set_index() function.it takes two parameters one is column name you want to change as index, and another one is inplace=true. When inplace=True is passed, the data is renamed inplace, when inplace=False is passed (this is the default value, so isn't necessary), performs the operation and returns a copy of the object.

```
In [358]: df.set_index('DateTime',inplace=True)

In [359]: df
```

Out[359]:

DateTime	QueueName	Q_Used	TotalMemoryGB	FreeMemoryGB	Processors	FreeMemoryPercentage
2018-05-23 15:42:00	6	0	1.38	0.31	2	22.57
2018-05-23 15:43:00	6	0	1.38	0.31	2	22.52
2018-05-23 15:44:00	6	0	1.38	0.30	2	21.51
2018-05-23 15:45:00	6	0	1.38	0.30	2	21.48
2018-05-23 15:46:00	6	0	1.38	0.29	2	21.25
2018-05-23 15:52:00	6	0	1.38	0.39	2	28.26
2018-05-23 15:53:00	6	0	1.38	0.39	2	28.23
2018-05-23 15:56:00	6	0	1.38	0.43	2	31.52
2018-05-23 15:57:00	6	0	1.38	0.43	2	31.47
2018-05-23 15:58:00	6	0	1.38	0.43	2	31.26
2018-05-23 16:01:00	6	0	1.38	0.32	2	23.46
2018-05-23 16:04:00	6	0	1.38	0.53	2	38.40
2018-05-23 16:06:00	6	0	1.38	0.59	2	42.87
2018-05-23 16:10:00	6	0	1.38	0.57	2	41.47
2018-05-23 16:11:00	6	0	1.38	0.55	2	39.60
2018-05-23 16:13:00	6	0	1.38	0.46	2	33.49
2018-05-23 16:14:00	6	0	1.38	0.38	2	27.69
2018-05-23 16:17:00	6	0	1.38	0.09	2	6.78

Now the DateTime is the index of the dataframe. Now we can perform DateTime operations very easily.

Data Frame Filtering based on index
How to filter data based on particular year.

To Check all the values occurred in a particular year let's say 2018
Run command df['2018'].

```
In [378]: df['2018']
Out[378]:
```

DateTime	QueueName	Q_Used	TotalMemoryGB	FreeMemoryGB	Processors	FreeMemoryPercentage
2018-06-22 15:42:00	6	0	1.58	0.31	2	22.57
2018-06-22 15:43:00	6	0	1.58	0.31	2	22.52
2018-06-22 15:44:00	6	0	1.58	0.30	2	21.51
2018-06-22 15:46:00	6	0	1.58	0.30	2	21.46
2018-06-22 15:48:00	6	0	1.58	0.29	2	21.26
2018-06-22 15:52:00	6	0	1.58	0.39	2	28.28
2018-06-22 15:53:00	6	0	1.58	0.39	2	28.23
2018-06-22 15:56:00	6	0	1.58	0.43	2	31.52
2018-06-22 15:57:00	6	0	1.58	0.43	2	31.47
2018-06-22 15:58:00	6	0	1.58	0.43	2	31.38
2018-06-22 16:01:00	6	0	1.58	0.52	2	33.46
2018-06-22 16:05:00	6	0	1.58	0.53	2	36.40
2018-06-22 16:06:00	6	0	1.58	0.59	2	42.87
2018-06-22 16:10:00	6	0	1.58	0.57	2	41.47
2018-06-22 16:11:00	6	0	1.58	0.55	2	39.60
2018-06-22 16:13:00	6	0	1.58	0.49	2	33.49
2018-06-22 16:14:00	6	0	1.58	0.36	2	27.80

Above result gives all the values recorded in year 2018.

How to filter data based on year and month

To View all observations that occurred in June 2018, run the below command

```
In [379]: df['2018-06']
Out[379]:
```

DateTime	QueueName	Q_Used	TotalMemoryGB	FreeMemoryGB	Processors	FreeMemoryPercentage
2018-06-24 10:34:00	6	0	1.37	0.08	2	6.87
2018-06-24 10:36:00	6	0	1.37	0.08	2	6.04
2018-06-24 10:36:00	6	0	1.37	0.46	2	33.86
2018-06-24 10:37:00	6	0	1.37	0.46	2	33.78
2018-06-24 10:38:00	6	0	1.37	0.12	2	8.48
2018-06-24 10:39:00	6	0	1.37	0.11	2	8.16
2018-06-24 10:40:00	6	0	1.37	0.10	2	7.16
2018-06-24 10:41:00	6	0	1.37	0.61	2	44.76
2018-06-24 10:42:00	6	4	1.37	0.21	2	15.19
2018-06-24 10:43:00	6	0	1.37	0.55	2	40.28
2018-06-24 10:44:00	6	0	1.37	0.20	2	14.76
2018-06-24 10:46:00	6	0	1.37	0.20	2	14.67
2018-06-24 10:46:00	6	0	1.37	0.17	2	12.67
2018-06-24 10:47:00	6	0	1.37	0.17	2	12.63
2018-06-24 10:48:00	6	3	1.37	0.24	2	17.36
2018-06-24 10:49:00	6	0	1.37	0.23	2	16.43
2018-06-24 11:00:00	6	0	1.37	0.33	2	24.37
2018-06-24 11:01:00	6	0	1.37	0.33	2	24.34
2018-06-24 11:02:00	6	0	1.37	0.32	2	23.17

Similarly, if you want to view the observations after particular year, month and date. then the command is

```
df[datetime(year, month, date):]
```

If you need observations between two dates then command is

```
df['Starting_year, Starting_month, Starting_date':'Ending_year, Ending_month, Ending_date']
```

For ex: if you need Observations between May 3rd and May 4th Of 2018
then command is: df['5/3/2018':'5/4/2018']

Pandas Indexing and Selecting Data

What is Indexing ?

Indexing in pandas means simply selecting particular rows and columns of data from a DataFrame. Indexing could mean selecting all the rows and some of the columns, some of the rows and all the columns, or some of each of the rows and columns. Indexing can also be known as **Subset Selection**.

Let's load one csv file and convert that to data frame to perform the indexing and selection operations.

```
In [465]: import pandas as pd
          df=pd.read_csv(r"C:\Users\SRIKARVARNAVALIVARTH\Desktop\Anaconda Notebook Jupyter\index.csv")
          df
```

Out[465]:

	Name	email	id	team
0	Judith	judith.diaz@sysiphus.com	200100	1
1	Victoria	victoria.price@sysiphus.com	200101	1
2	Leigh	leigh.snyder@sysiphus.com	200102	1
3	Rodney	rodney.barton@sysiphus.com	200103	1
4	Lucy	lucy.garza@sysiphus.com	200104	1
5	Fred	fred.clarke@sysiphus.com	200105	2
6	Jesse	jesse.parks@sysiphus.com	200106	2
7	Lamar	lamar.ruiz@sysiphus.com	200107	2
8	Eric	eric.gonzalez@sysiphus.com	200108	2
9	Arlene	arlene.hughes@sysiphus.com	200109	2

Once the data is loaded into data frame let's make Name as the index of this data frame.

Note: index is the primary key it should not contain duplicates

```
In [466]: df.set_index("Name",inplace=True)

In [467]: df
```

Out[467]:

Name	email	id	team
Judith	judith.diaz@sysiphus.com	200000	1
Victoria	victoria.price@sysiphus.com	200001	1
Leigh	leigh.snyder@sysiphus.com	200002	1
Rodney	rodney.barton@sysiphus.com	200003	1
Lucy	lucy.garza@sysiphus.com	200004	1
Fred	fred.clarke@sysiphus.com	200005	2
Jesse	jesse.parks@sysiphus.com	200006	2
Lamar	lamar.ruiz@sysiphus.com	200007	2
Eric	eric.gonzalez@sysiphus.com	200008	2
Arlene	arlene.hughes@sysiphus.com	200009	2

Selecting Single Column

In order to take single column, we simply put the name of the column in-between the

```
In [468]: # retrieving columns by indexing operator
          Email = df["email"]

In [469]: Email

Out[469]: Name
          Judith          judith.diaz@sysiphus.com
          Victoria      victoria.price@sysiphus.com
          Leigh           leigh.snyder@sysiphus.com
          Rodney        rodney.barton@sysiphus.com
          Lucy             lucy.garza@sysiphus.com
          Fred            fred.clarke@sysiphus.com
          Jesse           jesse.parks@sysiphus.com
          Lamar            lamar.ruiz@sysiphus.com
          Eric          eric.gonzalez@sysiphus.com
          Arlene        arlene.hughes@sysiphus.com
          Name: email, dtype: object
```

brackets.

Selecting Multiple Columns

To select multiple columns, we must pass a list of columns in an indexing operator.

```
In [470]: # retrieving multiple columns by indexing operator
          EmailANDTeam = df[["email","team"]]

In [471]: EmailANDTeam

Out[471]:
```

Name	email	team
Judith	judith.diaz@sysiphus.com	1
Victoria	victoria.price@sysiphus.com	1
Leigh	leigh.snyder@sysiphus.com	1
Rodney	rodney.barton@sysiphus.com	1
Lucy	lucy.garza@sysiphus.com	1
Fred	fred.clarke@sysiphus.com	2
Jesse	jesse.parks@sysiphus.com	2
Lamar	lamar.ruiz@sysiphus.com	2
Eric	eric.gonzalez@sysiphus.com	2
Arlene	arlene.hughes@sysiphus.com	2

Selecting a single row:

In order to select a single row using. loc[], we put a single row label in a .loc function.

```
In [474]: LucyData=df.loc["Lucy"]

In [475]: LucyData

Out[475]: email     lucy.garza@sysiphus.com
          id                         200004
          team                            1
          Name: Lucy, dtype: object
```

Selecting multiple rows:

In order to select multiple rows, we put all the row labels in a list and pass that to .loc function.

```
In [477]: jesseANDEric=df.loc[['Jesse','Eric']]

In [478]: jesseANDEric

Out[478]:
```

Name	email	id	team
Jesse	jesse.parks@sysiphus.com	200006	2
Eric	eric.gonzalez@sysiphus.com	200008	2

Selecting multiple rows and columns:

In order to select two rows and two columns, we select two rows which we want to select and two columns and put it in a separate list:
Dataframe.loc[["row1", "row2"], ["column1", "column2"]]

```
In [482]: selectedDF=df.loc[['Judith','Arlene'],['email','team']]

In [483]: selectedDF

Out[483]:
```

Name	email	team
Judith	judith.diaz@sysiphus.com	1
Arlene	arlene.hughes@sysiphus.com	2

In order to select all the rows and some columns the syntax looks like:

Dataframe.loc [:, ["column1", "column2"]]

```
In [487]: selectedDF=df.loc[: ,['email','team']]

In [488]: selectedDF

Out[488]:
```

Name	email	team
Judith	judith.diaz@sysiphus.com	1
Victoria	victoria.price@sysiphus.com	1
Leigh	leigh.snyder@sysiphus.com	1
Rodney	rodney.barton@sysiphus.com	1
Lucy	lucy.garza@sysiphus.com	1
Fred	fred.clarke@sysiphus.com	2
Jesse	jesse.parks@sysiphus.com	2
Lamar	lamar.ruiz@sysiphus.com	2
Eric	eric.gonzalez@sysiphus.com	2
Arlene	arlene.hughes@sysiphus.com	2

Pandas- groupby

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

Let’s load one csv file and convert that to data frame to perform the group-by operations.

```
In [499]: import pandas as pd
          df=pd.read_csv(r"C:\Users\SRIKARVARNAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\groups.csv")
          df
```

Out[499]:

	hlpi_name	year	own_prop	own_ent_prop	prop_hhs	age	size	income	expenditure	eqv_income	eqv_exp
0	All households	2003	69.7	36.0	100.0	35.9	2.7	40794	42264	26669	25132
1	Beneficiary	2003	30.3	21.2	11.9	29.9	2.8	23404	25270	14258	15024
2	Income quintile 1 (low)	2003	61.5	15.5	20.0	40.8	2.3	10747	21145	13402	14489
3	Income quintile 2	2003	62.8	26.9	20.0	34.7	2.8	31300	29655	18917	18299
4	Income quintile 3	2003	69.7	46.3	20.0	31.5	3.0	49106	46561	26878	24972
5	Income quintile 4	2003	73.3	47.3	20.0	35.3	2.6	61674	52776	36691	31963
6	Income quintile 5 (high)	2003	81.3	49.1	20.0	39.3	2.5	96861	72822	55637	42932
7	Expenditure quintile 1 (low)	2003	62.1	16.8	20.0	38.7	2.5	23980	16413	16190	16315
8	Expenditure quintile 2	2003	66.1	27.7	20.0	36.1	2.7	34155	29685	20357	18121
9	Expenditure quintile 3	2003	62.3	34.7	20.0	33.9	2.8	49771	42462	27203	25132
10	Expenditure quintile 4	2003	74.2	47.7	20.0	35.1	2.7	60563	59615	34547	34187
11	Expenditure quintile 5 (high)	2003	83.6	56.2	20.0	36.7	2.5	77434	89653	46269	51550
12	Maori	2003	47.4	30.5	14.2	28.9	3.2	42585	35312	23096	19797
13	Superannuitant	2003	87.6	5.1	19.2	70.3	1.6	22367	21536	17203	17211
14	All households	2011	65.2	32.8	100.0	38.3	2.6	53103	48058	30633	27335
15	Beneficiary	2011	29.7	13.9	12.3	29.9	2.7	25902	27665	16097	15385
16	Income quintile 1 (low)	2011	51.7	15.5	20.0	36.3	2.4	19787	24224	15414	13221
17	Income quintile 2	2011	58.2	24.1	20.0	35.9	2.9	37370	34260	21938	20383
18	Income quintile 3	2011	63.0	37.3	20.0	33.4	2.9	54894	46431	30033	25130
19	Income quintile 4	2011	70.6	41.5	20.0	38.8	2.6	69183	65560	42084	33319
20	Income quintile 5 (high)	2011	81.0	44.8	20.0	42.9	2.4	106227	71815	63106	44712
21	Expenditure quintile 1 (low)	2011	53.9	11.2	20.0	37.3	2.6	27501	16877	17612	13377
22	Expenditure quintile 2	2011	55.7	23.9	20.0	35.1	2.7	38932	32750	22895	20168
23	Expenditure quintile 3	2011	65.9	33.8	20.0	35.3	2.8	56117	46651	32053	27336

groupby function based on single category

Now we have data frame ready let’s group the data based on ‘hlpi_name’.

```
In [501]: groups = df.groupby('hlpi_name')

In [502]: groups
Out[502]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001737DCC1F98>
```

Once group by operation is done we get a result as groupby object.

Let’s print the value contained in any one of group. For that use the name of the ‘hlpi_name’. We use the function get_group() to find the entries contained in any of the groups.

```
In [514]: #To know all the unique values(groups) in hlpi_name
          df['hlpi_name'].drop_duplicates()

Out[514]: 0                    All households
          1                       Beneficiary
          2            Income quintile 1 (low)
          3                 Income quintile 2
          4                 Income quintile 3
          5                 Income quintile 4
          6           Income quintile 5 (high)
          7       Expenditure quintile 1 (low)
          8             Expenditure quintile 2
          9             Expenditure quintile 3
          10            Expenditure quintile 4
          11     Expenditure quintile 5 (high)
          12                             Maori
          13                     Superannuitant
          Name: hlpi_name, dtype: object
```

```
In [519]: #To get values contained in any one of group
          groups.get_group('Income quintile 1 (low)')
```

Out[519]:

	year	own_prop	own_wm_prop	prop_hhs	age	size	income	expenditure	eqv_income	eqv_exp
2	2008	61.3	15.5	20.0	40.0	2.3	16747	21145	13402	14408
16	2011	51.7	15.5	20.0	38.3	2.4	19787	24224	15414	16221
30	2014	52.1	16.0	20.0	37.4	2.4	22822	25809	17188	17555
44	2017	54.1	12.9	20.0	40.3	2.3	22733	26775	18859	17850

groupby function based on more than one category

Use groupby() function to form groups based on more than one category (i.e. Use more than one column to perform the splitting).

```
In [526]: group_2 = df.groupby(['hlpi_name', 'year'])
          group_2

Out[526]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001737600D9080>
```

We got the result as groupby object.

Let’s print the first entries in all the groups formed using first() function.

```
In [526]: group_2 = df.groupby(['hlpi_name', 'year'])
          group_2
```

```
Out[526]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x000000[illegible]>
```

```
In [527]: group_2.first()
```

Out[527]:

		own_prop	own_wm_prop	prop_hhs	age	size	income	expenditure	eqv_income	eqv_exp
hlpi_name	year									
All households	2008	69.7	36.6	100.0	35.9	2.7	46704	42594	26889	25152
	2011	69.2	32.6	100.0	36.3	2.6	53103	46098	30833	27339
	2014	66.5	33.7	100.0	37.0	2.6	57359	48338	33426	29683
	2017	66.6	33.0	100.0	37.4	2.7	64098	54293	36148	31489
Beneficiary	2008	38.5	21.2	11.9	39.9	2.6	25404	25270	14258	15824
	2011	28.7	13.8	12.1	38.0	2.7	25402	27685	16097	16685
	2014	24.6	14.5	11.4	37.3	2.5	26559	27348	17706	17895
	2017	22.8	10.8	7.8	39.1	2.8	29947	30054	16822	17655
Expenditure quintile 1 (low)	2008	62.1	15.8	20.0	38.7	2.5	23880	16413	15150	11915
	2011	53.9	11.2	20.0	37.3	2.6	27501	18877	17612	13877
	2014	54.9	12.7	20.0	38.5	2.6	30138	19997	19171	13678
	2017	53.8	9.7	20.0	40.0	2.6	33596	20167	20674	14437
Expenditure quintile 2	2008	66.1	27.7	20.0	36.1	2.7	34155	29085	20357	18121
	2011	65.7	23.0	20.0	36.1	2.7	38932	32799	22858	20103
	2014	58.4	24.3	20.0	36.7	2.7	42908	35822	26743	21961
	2017	67.6	23.7	20.0	36.1	2.8	50681	38952	27694	22792

Operations on groups

After splitting a data into a group, we can also apply a function to each group to perform some operations.
Here is the Sample example to get sum of values in particular groups.

```
In [540]: import pandas as pd
          df=pd.read_csv(r"C:\Users\[illegible]\Desktop\Anaconda-Notebook-Jupyter\groups.csv")
          Group = df.groupby(['hlpi_name'])
          Group.sum()
```

Out[540]:

	year	own_prop	own_wm_prop	prop_hhs	age	size	income	expenditure	eqv_income	eqv_exp
hlpi_name										
All households	8050	268.0	130.1	400.0	146.0	10.6	221202	162115	127274	103559
Beneficiary	8050	114.8	60.5	45.4	154.3	10.6	105820	113777	64803	68657
Expenditure quintile 1 (low)	8050	224.7	59.4	80.0	154.5	10.3	114915	79464	72647	52267
Expenditure quintile 2	8050	237.0	93.6	80.0	142.0	10.9	139546	135740	95503	82642
Expenditure quintile 3	8050	259.0	137.6	80.1	140.4	11.3	251377	195579	135356	113582
Expenditure quintile 4	8050	288.1	177.4	80.0	144.0	10.6	297302	250114	199203	152697
Expenditure quintile 5 (high)	8050	350.4	210.6	80.0	151.9	10.0	571471	392890	228301	226569
Income quintile 1 (low)	8050	219.2	59.9	80.0	154.0	9.4	62089	97953	64648	66634
Income quintile 2	8050	235.1	100.3	80.0	139.1	11.0	150400	149220	99892	85672
Income quintile 3	8050	264.7	150.5	80.0	134.4	11.8	229555	209866	127275	113843
Income quintile 4	8050	298.3	176.6	80.0	147.4	10.5	296802	242338	177382	143243
Income quintile 5 (high)	8050	338.9	197.4	80.0	162.3	9.8	439239	313420	279583	191459
Maori	8050	183.1	119.2	66.6	119.4	12.6	213329	173677	113318	93693
Superannuitant	8050	344.0	32.5	84.1	278.7	8.5	117510	106788	88745	84782

We can also find min, max, average . .etc.

Merge/Join Datasets

Joining and merging DataFrames is the core process to start with data analysis and machine learning tasks. It is one of the toolkits which every Data Analyst or Data Scientist should master because in almost all the cases data comes from multiple source and files. You may need to bring all the data in one place by some sort of join logic and then start your analysis. Thankfully you have the most popular library in python, pandas to your rescue! Pandas provides various facilities for easily combining different datasets.

We can merge two data frames in pandas python by using the merge() function. The different arguments to merge() allow you to perform natural join, left join, right join, and full outer join in pandas.

Understanding the different types of merge:

Before you perform joint operations let's first load the two csv files and convert them into data frames df1 and df2.

```
In [541]: import pandas as pd
          df1=pd.read_csv(r"C:\Users\SRICARVARNAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\Joints\df1.csv")
          df1
```

Out[541]:

	customer_id	Product
0	1	Oven
1	2	Television
2	3	AC
3	4	Washing Machine
4	5	AC
5	6	Oven
6	7	Television
7	8	Washing Machine
8	9	Television
9	10	Washing Machine

```
In [542]: df2=pd.read_csv(r"C:\Users\SRICARVARNAVALIVARTH\Desktop\Anaconda-Notebook-Jupyter\Joints\df2.csv")
          df2
```

Out[542]:

	customer_id	state
0	1	Texas
1	2	California
2	4	Florida
3	7	California
4	10	Florida

Natural join

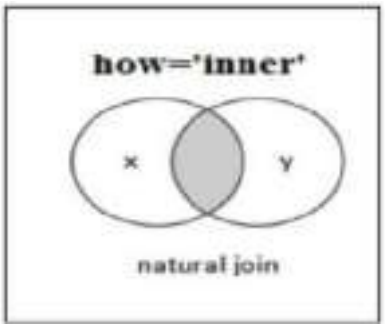
Natural join keeps only rows that match from the data frames(df1 and df2), specify the

argument how=’inner’

Syntax:

pd.merge(df1, df2, on=column', how='inner')

Return only the rows in which the left table have matching keys in the right table



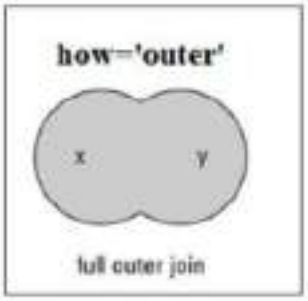

```
In [545]: pd.merge(df1, df2, on='customer_id', how='inner')
```

Out[545]:

	customer_id	Product	state
0	1	Oven	Texas
1	2	Television	California
2	4	Washing Machine	Florida
3	7	Television	California
4	10	Washing Machine	Florida

Full outer join

Full outer join keeps all rows from both data frames, specify how=‘outer’.



Syntax:

pd.merge(df1, df2, on=column', how=’outer’)

Returns all rows from both tables, join records from the left which have matching keys in the right table.

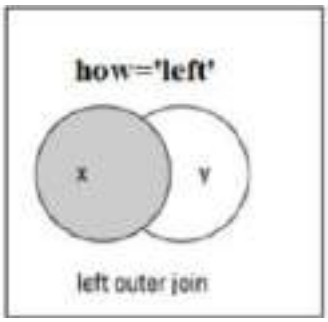
```
In [546]: pd.merge(df1, df2, on='customer_id', how='outer')
```

Out[546]:

	customer_id	Product	state
0	1	Oven	Texas
1	2	Television	California
2	3	AC	NaN
3	4	Washing Machine	Florida
4	5	AC	NaN
5	6	Oven	NaN
6	7	Television	California
7	8	Washing Machine	NaN
8	9	Television	NaN
9	10	Washing Machine	Florida

Left outer join

Left outer join includes all the rows of your data frame df1 and only those from df2 that match, specify how =‘Left.



Syntax:

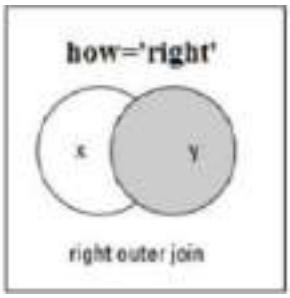
pd.merge(df1, df2, on=column', how=left)
Return all rows from the left table, and any rows with matching keys from the right table.

```
In [547]: pd.merge(df1, df2, on='customer_id', how='left')
Out[547]:
```

	customer_id	Product	state
0	1	Oven	Texas
1	2	Television	California
2	3	AC	NaN
3	4	Washing Machine	Florida
4	5	AC	NaN
5	6	Oven	NaN
6	7	Television	California
7	8	Washing Machine	NaN
8	9	Television	NaN
9	10	Washing Machine	Florida

Right outer join

Return all rows from the df2 table, and any rows with matching keys from the df1 table, specify how =‘Right’.



Syntax:

pd.merge(df1, df2, on=column', how=right)
Return all rows from the right table, and any rows with matching keys from the left table.

```
In [548]: pd.merge(df1, df2, on='customer_id', how='right')
Out[548]:
```

	customer_id	Product	state
0	1	Oven	Texas
1	2	Television	California
2	4	Washing Machine	Florida
3	7	Television	California
4	10	Washing Machine	Florida