Chapter-5 : Python Collections and Library

Python Collections (container data types)

The collection Module in Python provides different types of containers. A Container is an object that is used to store different objects and provide a way to access the contained objects and iterate over them. Some of the built-in containers are Tuple, List, Dictionary and set.

→ Tuples in Python

Tuples are used to store multiple items in a single variable. Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage. A tuple is a collection which is ordered and unchangeable.

Tuple properties:

- Ordered: When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.
- Unchangeable: Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.
- Allow Duplicates: Since tuple are indexed, tuples can have items with the same value
- Indexed: the first item has index [0], the second item has index [1] etc.

Declaring tuples: Tuples are declared with round brackets.

```
tuple1 = ("one","two","three")
print(tuple1)
O/P: ("one","two","three")
```

Tuple elements are indexed starting from 0. So we can access individual elements of a tuple using square brackets and index, e.g. tuple1[0], the way we access array elements. Also Python provides special for loop to iterate through all container objects like tuple. tuples allow duplicate values. See following code that illustrates use of index to access tuple elements and use of for loop to iterate through tuple.

```
tuple1 = ("one","two","three","two","one")  #duplicate values
print(tuple1)

print("iterating tuple with index & while loop")
i=0
while i < len(tuple1):
      print(tuple1[i])
      i = i + 1

print("now iterating tuple with for loop")
for x in tuple1:      #iterating tuple with for loop
      print(x)
O/P:
('one', 'two', 'three', 'two', 'one')
iterating tuple with index & while loop
one
two
```

```
three
two
one
now iterating tuple with for loop
one
two
three
two
one
```

len( ) function can be used to find the length (total number of elements) of a tuple, as shown in the above example.

A tuple can contain **different data types**, for example a tuple with strings, integers and boolean values:

tuple1 = ("abc", 34, True, 40, "male")

The tuple() constructor / method
It is also possible to use the tuple() constructor to create a tuple.

```
tuple1 = tuple(("one","two","three"))
print(tuple1)
print(type(tuple1))
O/P:
("one","two","three")
<class 'tuple'>
```

Changing tuple values, adding into or removing data from tuple is not possible once a tuple is created. That is its property and that's why tuples are called immutable or unchangeable.

count( ) method can be used with a tuple to count number of times an element occurs in a tuple.

Syntax: tuple.count(element)

```
tuple1 = (10,20,10,30,10,40)
x = tuple1.count(10)
print("count of 10 is:",x)
O/P:
Count of 10 is: 3
```

The index() method finds the first occurrence of the specified value. it raises an exception if the value is not found.

Syntax: tuple.index(element)

```
tuple1 = (10,20,10,30,10,20)
x = tuple1.index(20)
print("index of 20 is:",x)
O/P:
index of 10 is: 1
```

→ Sets in Python

Sets are used to store multiple items in a single variable. Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage. A set is a collection which is both unordered and un-indexed. Sets are written with curly brackets.

Tuple properties:

- unordered: When we say that sets are unordered, it means that the items appear in random order every time we use the set.
- Un-indexed: The elements of set are un-indexed.
- Changeable (mutable): Sets are mutable, that means we can change elements of a set once it is created. But not using index because sets are un-indexed.
- Ignores Duplicates: Sets allows duplicate item entries, but duplicates are ignored.

Declaring sets: sets are declared with curly brackets.

```
set1 = {"one","two","one"}
print(set1)
O/P: {"one","two"}
```

A set can contain elements of **different data types**.

```
set1 = {"one","two","three"}
set2 = {1, 3, 5, 7, 9}
set3 = {True, False, True}
set4 = {"one",22,True,40.5,"Surat"}
```

The set() constructor / method

It is also possible to use the set( ) constructor to create a set.

```
set1 = set({"Sun","Mon","Tue"})
print(set1)
print(type(set1))
O/P:
{"Sun","Mon","Tue"}
<class 'set'>
```

Note that sets are not indexed, so we cannot access its elements using while loop and index as we can do with tuples. But we can still use for loop to iterate through a set. Following sample code shows how to.

```
set1 = {"Mon","Tue","Wed","Thu","Fri"}
print(set1)

print("now iterating set with for loop")
for x in set1:        #iterating set with for loop
     print(x)
O/P:
{'Thu', 'Fri', 'Mon', 'Tue', 'Wed'}
now iterating set with for loop
Thu
```

```
Fri
Mon
Tue
Wed
```

len( ) function can be used to find the length (total number of elements) of a set.

count( ) method is not supported by set because set ignores duplicates, so any element if present in a set has only one occurrence.

index( ) method is not supported by set because sets are un-index container data types.

Set operations available in Python:

This data structure set in Python is created based on set theory of Mathematics. We all have studies different set theory operations like union and intersection. In Python, some are performed by operators, some by methods, and some by both.

The general rule is that: where a set is expected, methods will typically accept any element as an argument, but operators require actual sets as operands**.**

Union: Union of two or more than two sets can be done using either union( ) method or by pipe (|) operator.

```
x1 = {'a', 'b', 'c'}
x2 = {'c', 'd', 'e'}
x3 = x1.union(x2)
print(x3)
x3 = x1 | x2
print(x3)
a = {1, 2, 3, 4}
b = {2, 3, 4, 5}
c = {3, 4, 5, 6}
d = a.union(b, c)
print(d)
d = a | b | c
print(d)
```

O/P:

```
{'a', 'b', 'c', 'd', 'e'}
{'a', 'b', 'c', 'd', 'e'}
{1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5, 6}
```

Intersection: Intersection of two or more than two sets can be done using either intersection() method or by ampersand (&) operator.

```
x1 = {'a', 'b', 'c'}
x2 = {'c', 'd', 'e'}
x3 = x1.intersection(x2)
print(x3)
x3 = x1 & x2
print(x3)
a = {1, 2, 3, 4}
b = {2, 3, 4, 5}
```

```
c = {3, 4, 5, 6}
d = a.intersection(b, c)
print(d)
d = a & b & c
print(d)
O/P:
{'c'}
{'c'}
{3, 4}
{3, 4}
```

The set add() method adds a given element to a set if the element is not present in the set. The add() method doesn't add an element to the set if it's already present in it otherwise it will get added to the set.

add() takes single parameter(element) which needs to be added in the set. The add() method doesn't return any value.

```
a = {1, 2, 3, 4}
a.add(5)
print(a)        #O/P: {1, 2, 3, 4, 5}
```

The clear() method can be used with a set object to clear (remove) all elements from the set.

```
a = {1, 2, 3, 4}
a.clear()       #clears the set
print(a)
```

The copy() method can be used with a set object to create a copy of the set. The method, when used, returns a copy of the set.

```
a = {1, 2, 3, 4}
b = a.copy()
print(b)        #O/P: {1, 2, 3, 4}
```

The remove() and discard() methods are used with a set object to remove a specified element from the set.

Syntax: set.remove(element)
set.discard(element)

The difference between remove and discard is that; when the specified element is not found in the set, the remove( ) method will raise an error, while discard( ) method will not.

```
a = {1, 2, 3, 4}
a.remove(3)
a.discard(3)
print(a)        #O/P: {1, 2, 4}
```

The pop() method removes a random item from the set. This method returns the removed item. Note that sets are unordered and un-indexed, that's why removes a random item.

```
a = {1, 2, 3, 4}
x = a.pop()
print(x)
print(a)
```

O/P:

```
2
{1, 3, 4}
```

The update() method in set adds elements from one set (passed as an argument) to another set. Update() method takes only a single argument. The single argument can be a set, list, tuple or a dictionary. It automatically converts into a set and adds to the set. This method returns nothing.

```
x1 = {'a', 'b', 'c'}
x2 = {'c', 'd', 'e'}
x1.update(x2)
print(x1)
```

O/P:

```
{'c', 'd', 'a', 'e', 'b'}
```

→ Dictionary in Python

Dictionary in Python is an unordered collection of data values, used to store data values like a map. Unlike other Python data types that hold only single value as an element, dictionary holds **key : value pair**. Key value is provided in the dictionary to make it more optimized.

Creating a Dictionary

In Python, a Dictionary can be created by placing sequence of elements within curly {} braces, separated by 'comma'. Dictionary holds a pair of values, one being the Key and the other corresponding pair element being its key : value.

- Values in a dictionary can be of any data type and can be duplicated (values only).
- Keys can't be repeated and are immutable, means can't be changed later.
- Dictionary keys are case sensitive, same name but different cases of Key will be treated differently.
- Dictionary items are ordered.
- Dictionary items are presented in key : value pairs, and can be referred to by using the key name.
- Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.
- Dictionaries cannot have two items with the same key (no duplicate keys).

Creating a Dictionary

```
# with Integer Keys
dict1 = {1: 'Surat', 2: 'Bharuch', 3: 'Vadodara'}
print("Dictionary with Integer Keys: ")
print(dict1)
# with Mixed keys
dict2 = {'name': 'Vishal', 1: [11, 22, 33]}
print("Dictionary with Mixed Keys: ")
print(dict2)
print("key name=",dict2['name'])      #get element using key
```

O/P:

```
Dictionary with Integer Keys:
{1: 'Surat', 2: 'Bharuch', 3: 'Vadodara'}
Dictionary with Mixed Keys:
{'Name': 'Vishal', 1: [11, 22, 33]}
key name= Vishal
```

Adding key : value pairs to a dictionary

Key : value pairs can be added to an existing dictionary by using a key as an index. One value at a time can be added to a Dictionary by defining value along with the key e.g. Dict[Key] = 'Value'.

```
dict1 = {1: 'Surat', 2: 'Bharuch', 3: 'Vadodara'}
print(dict1)
print("After adding a new key:value pair")
dict1[4] = 'Rajkot'
print(dict1)
```

```
O/P:
{1: 'Surat', 2: 'Bharuch', 3: 'Vadodara'}
After adding a new key:value pair
{1: 'Surat', 2: 'Bharuch', 3: 'Vadodara', 4: 'Rajkot'}
```

Removing Items from a Dictionary

There are several methods to remove items from a dictionary.

The pop() method removes the item with the specified key name.

```
dict1 = {1: 'Surat', 2: 'Bharuch', 3: 'Vadodara'}
print(dict1)
dict1.pop(2)
print("After popping element with key:2")
print(dict1)
```

```
O/P:
{1: 'Surat', 2: 'Bharuch', 3: 'Vadodara'}
After popping element with key:2
{1: 'Surat', 3: 'Vadodara'}
```

The popitem() method removes the last inserted item.

```
dict1 = {1: 'Surat', 2: 'Bharuch', 3: 'Vadodara'}
dict1.popitem()
```

The del keyword removes the item with the specified key name.

```
dict1 = {1: 'Surat', 2: 'Bharuch', 3: 'Vadodara'}
del dict1[2]
```

The del keyword can also delete the dictionary completely.

```
dict1 = {1: 'Surat', 2: 'Bharuch', 3: 'Vadodara'}
del dict1
```

The clear() method clears (removes all items) the dictionary.

```
dict1 = {1: 'Surat', 2: 'Bharuch', 3: 'Vadodara'}
dict1.clear()
```

They copy() method returns a copy of the dictionary. The copy() method doesn't take any parameters. This method doesn't modify the original dictionary just returns copy of the dictionary.

```
dict1 = {1: 'Surat', 2: 'Bharuch', 3: 'Vadodara'}
dict2 = dict1.copy()
```

Introduction to Numpy and Pandas

→ Numpy (Numerical Python)

NumPy is a Python library that provides a simple yet powerful data structure: the n-dimensional array. Learning Numpy is said to be the first step towards data science in Python. This is the foundation on which almost all the power of Python's data science toolkit is built, and learning NumPy is the first step on any Python data scientist's journey. Apart from functions for arrays, Numpy also provides functions for working in linear algebra and matrices. NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

Advantage of using Numpy:
In Python we have lists that serve the purpose of arrays, but they are slow to process. NumPy aims to provide an array object that is up to 50x faster than traditional Python lists. The array object in NumPy is called ndarray (n dimensional array), it provides a lot of supporting functions that make working with ndarray very easy. Arrays are very frequently used in data science, where speed and resources are very important.

Ndarrays of Numpy are faster than lists because, NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently. Also Numpy is optimized to work with latest CPU architectures.

NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

Using Numpy ndarray

First we need to install this package named Numpy on our computer. If we have Python and PIP installed on our machine, then we can install Numpy using "pip install <package_name>" command.

Once successfully installed, to use any data structure or any methods from Numpy package, we have to first import it in our program using "import" keyword.

array( ) method: NumPy is used to work with arrays. The array object in NumPy is called ndarray. We can create a NumPy ndarray object by using the array() function. To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray. See following code for more understanding.

```
import numpy as np
# np is an alias given using 'as' keyword
a1 = np.array([11, 22, 33, 44])
print(a1)
print(type(arr))
O/P:
[11 22 33 44]
<class 'numpy.ndarray'>
```

Multi dimensional arrays: To create multi dimensional arrays in Numpy, we just create array of arrays, and array of array of arrays and so on. See following code for understanding.

```
#creating a 2D array
```

```
import numpy as np
a1 = np.array([[11, 22], [33, 44]])
print(a1)
O/P:
[[11 22]
 [33 44]]
```

In above example we have created an array which has two elements, which are 1D arrays themselves. So the array a1 became a 2D array.

→ Numpy methods

We are going to see some Numpy methods for some widely used statistical functions like;

- Mean - The average value
- Median - The mid-point value
- Mode - The most common value

1. numpy.mean( )

It is used to compute the arithmetic mean (average) of the given data (array elements) along the specified axis.

Syntax: numpy.mean(arr, axis = value)

arr = input array

axis = (optional part of syntax)

value = None means flat array,

0 means working along the column,

1 means working along the row. Default is None.

See following simple example without the use of axis.

```
import numpy as np
# using keyword 'as', numpy is given an alias np
a1 = [2, 22, 17, 11, 4]
print("a1:", a1)
print("mean of a1 is:", np.mean(a1))
O/P:
a1: [2, 22, 17, 11, 4]
mean of a1 is: 11.0
```

2. numpy.median( )

It is used to compute the median of the given data (array elements) along the specified axis.

How do we calculate median?

- Arrange given data elements in ascending order
- Median = middle term if total no. of terms are odd.
- Median = Average of the terms in the middle (if total no. of terms are even)

Syntax: numpy.mean(arr, axis = value)

```
import numpy as np
a1 = [2, 22, 17, 11, 4]
print("a1:", a1)
print("median of a1 is:", np.median(a1))
O/P:
```

```
a1: [2, 22, 17, 11, 4]
median of a1 is: 11.0
```

3. scipy.stats.mode( )

Numpy doesn't have a method to find mode. So we use mode( ) method from **scipy (Scientific Python)** package. It is used to compute the mode of the given data (array elements) along the specified axis. The Mode value is the value that appears the most number of times in the data set.

```
import scipy
from scipy import stats
a1 = [2, 22, 17, 22, 4]
print("a1:", a1)
print("mode of a1 is:", scipy.stats.mode(a1))
O/P:
a1: [2, 22, 17, 22, 4]
mode of a1 is: ModeResult(mode=array([22]), count=array([2]))
```

Understanding the result:

mode=array([22]) says that mode is 22.

count=array([2]) says that 22 appeared 2 times in the data set.

4. numpy.std( )

Numpy has std( ) method to calculate **standard deviation** from given data set. The method is used to calculate standard deviation of the given data (array elements) along the specified axis(if any).

```
import numpy
a1 = [2, 44, 3, 11, 5]
print("a1:", a1)
print("Standard deviation for a1 is:", numpy.std(a1))
O/P:
a1: [2, 44, 3, 11, 5]
Standard deviation for a1 is: 15.811388300841896
```

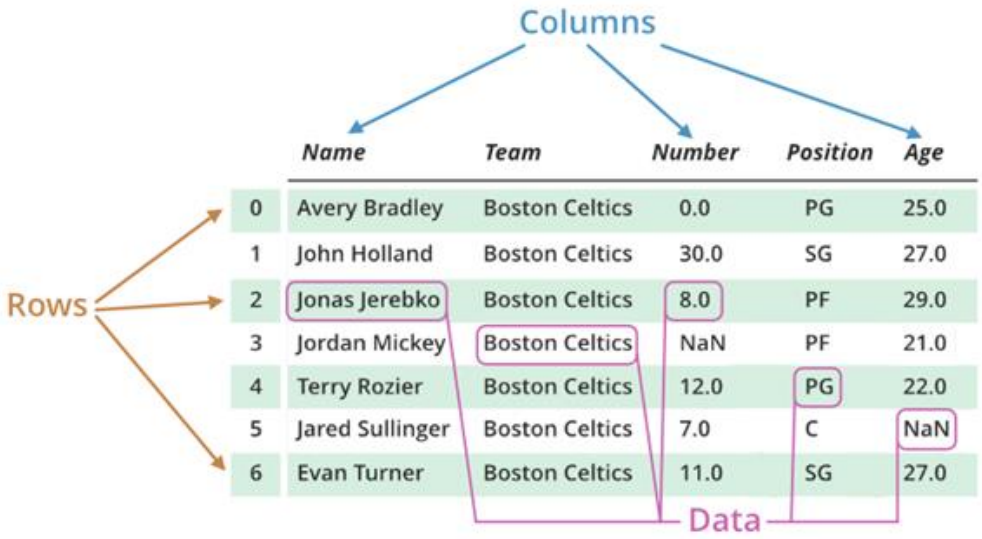
5. numpy.var( )

Numpy has var( ) method to calculate **variance** from given data set. The method is used to calculate variance from the given data (array elements) along the specified axis(if any).

```
import numpy
a1 = [2, 44, 3, 11, 5]
print("a1:", a1)
print("Variance for a1 is:", numpy.var(a1))
a1: [2, 44, 3, 11, 5]
Variance for a1 is: 250.0
```

→ Pandas Dataframe

Pandas dataFrame is a two dimensional tabular data structure which has labeled rows and columns. Data is stored here in a table form using rows and columns. Pandas DataFrame consists of three principal components, the data, rows, and columns.



	Name	Team	Number	Position	Age
0	Avery Bradley	Boston Celtics	0.0	PG	25.0
1	John Holland	Boston Celtics	30.0	SG	27.0
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN
6	Evan Turner	Boston Celtics	11.0	SG	27.0

In the real world, a Pandas DataFrame will be created by loading the datasets from existing storage, storage can be SQL Database, CSV file, and Excel file. Pandas DataFrame can be created from the lists, dictionary, and from a list of dictionary etc. Dataframe can be created in different ways, we will discuss some of the ways here.

Create a simple Pandas DataFrame

```
import pandas
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
df = pandas.DataFrame(data)
print(df)
```

O/P:

```
     calories  duration
  0       420        50
  1       380        40
  2       390        45
```

Creating a dataFrame using List

DataFrame can be created using a single list or a list of lists. We can use DataFrame( ) method available in pandas package. The constructor method returns a dataframe object.

Syntax: pandas.DataFrame(list)

```
import pandas
list1 = ['pandas', 'package', 'for', 'data', 'frame']
# Calling DataFrame constructor on list
df = panda.DataFrame(list1)
print(df)
```

```
O/P:
         0
0   pandas
1  package
2      for
3     data
4    frame
```

Creating dataframe using dict of equal length list

There are various ways of creating a DataFrame in Pandas. One way is to convert a dictionary containing lists of equal lengths as values.

Suppose we have a dictionary which contains product name as keys and list of top three selling cities as values.

```
import pandas
topSales = {'TV': ['Surat', 'Rajkot', 'Ahmedabad'],
            'laptop': ['Ahmedabad', 'Vadodara', 'Surat'],
            'AC': ['Rajkot', 'Vapi', 'Anand']}
# Convert the dictionary into DataFrame
topSales_pd = pandas.DataFrame(topSales)
print(topSales_pd)
```

```
O/P:
           TV     laptop      AC
0      Surat  Ahmedabad  Rajkot
1     Rajkot   Vadodara    Vapi
2  Ahmedabad      Surat   Anand
```

Reading data using csv file (read_csv( ))

Pandas is one of those packages and makes importing and analyzing data much easier. read_csv( ) is an important pandas function to read csv files and do operations on it.

Syntax: pd.read_csv("filename.csv")

But there are many others thing one can do through this function only to change the returned object completely. For instance, one can read a csv file not only locally, but from a URL through read_csv( ) or one can choose what columns needed to export so that we don't have to edit the array later.

Retrieving rows and columns from dataframe using index

Indexing is also known as Subset selection. It means to select Rows & Columns by Name or Index in Pandas DataFrame using square brackets [ ]. Indexing in Pandas means selecting rows and columns of data from a Dataframe. Selection could be;

- all the rows and specific columns,
- Specific rows and all the columns,
- Specific rows and specific columns.

Let's create a simple dataframe with a list of tuples. With column names: 'Name', 'Age', 'Class' and 'percentage'.

```
import pandas
students = [('Amar', 17, 'FYBCA', 75),
           ('Akbar', 18, 'FYBCA', 86),
           ('Anthoni', 18, 'SYBCA ', 67),
           ('Joli', 19, 'SYBCA', 81),
           ('Viru', 19, 'TYBCA', 77),
           ('Jay', 20, 'TYBCA', 83)]
df=
pandas.DataFrame(students,columns=['Name','Age','Class','Percent'])
print(df)
print("---------------Selected columns result")
result = df[['Name','Percent']]
print(result)
```

O/P:

```
      Name  Age   Class  Percent
0     Amar   17   FYBCA       75
1    Akbar   18   FYBCA       86
2  Anthoni   18  SYBCA        67
3     Joli   19   SYBCA       81
4     Viru   19   TYBCA       77
5      Jay   20   TYBCA       83
--------------- Selected columns result
      Name  Percent
0     Amar       75
1    Akbar       86
2  Anthoni       67
3     Joli       81
4     Viru       77
5      Jay       83
```

Using Dataframe.loc[ ]

The dataframe.loc[] function selects the data by labels of rows or columns. It can select a subset of rows and columns. There are many ways to use this function.

Let's use the same dataset students that we created in above example. Now suppose I want to locate record (row) of student named "Joli" from the data set, it is possible with following code.

```
result=df.loc['Joli']
```

But before we can do that we have to set "Name" column as index to locate (search) records from. Now this we can do using following code.

```
df.set_index("Name", inplace = True)
```

Here the concept of index is used. Before searching we have to set the column to which the data item that we want to use to search (locate) a record as index column. Then we can use using datafram.loc[ ] method. See following code with output for better understanding.

```
import pandas
students = [('Amar', 17, 'FYBCA', 75),
           ('Akbar', 18, 'FYBCA', 86),
           ('Anthoni', 18, 'SYBCA ', 67),
```

```
          ('Joli', 19, 'SYBCA', 81),
          ('Viru', 19, 'TYBCA', 77),
          ('Jay', 20, 'TYBCA', 83)]
df=
pandas.DataFrame(students,columns=['Name','Age','Class','Percent'])

df.set_index("Name", inplace = True)
result=df.loc['Joli']
print(result)

print("---------------selecting multiple rows")
result=df.loc[['Jay','Viru']]
print(result)

print("-----------selecting multiple rows with specific columns")
result = df.loc[['Jay','Viru'],['Class','Percent']]
print(result)

print("-----------all rows (:) with specific columns")
result = df.loc[:,'Class','Percent']]
print(result)
```

O/P:

```
Age           19
Class      SYBCA
Percent       81
Name: Joli, dtype: object
---------------selecting multiple rows
      Age  Class  Percent
Name
Jay    20  TYBCA       83
Viru   19  TYBCA       77
-----------selecting multiple rows with specific columns
      Class  Percent
Name
Jay   TYBCA       83
Viru  TYBCA       77
-----------all rows (:) with specific columns
          Class  Percent
Name
Amar      FYBCA       75
Akbar     FYBCA       86
Anthoni   SYBCA        67
Joli      SYBCA       81
Viru      TYBCA       77
Jay       TYBCA       83
```

loc( ) is label based data selecting method which means that we have to pass the name of the row or column which we want to select, as we saw in previous example.

Apart from that with loc( ) method, we can also select data based on specific conditions and we can slice Pandas dataframe by passing row range to loc( ) method. We will see this in following example code. Lets first create a dataframe.

```
import pandas

df = pandas.DataFrame({'Brand':['Maruti', 'Hyundai', 'Tata',
'Mahindra', 'Maruti', 'Hyundai', 'Renault', 'Tata', 'Maruti'],
'Year':[2012, 2014, 2011, 2015, 2012, 2016, 2014, 2018, 2019],
'Kms':[50000, 30000, 60000, 25000, 10000, 46000, 31000, 15000,
12000],
'City' : ['Gurgaon', 'Delhi', 'Mumbai', 'Delhi', 'Mumbai', 'Delhi',
'Mumbai','Chennai', 'Ghaziabad'],
'Mileage' : [28, 27, 25, 26, 28, 29, 24, 21, 24]})

display(df)
```

O/P:

```
      Brand  Year    Kms       City  Mileage
0    Maruti  2012  50000    Gurgaon       28
1   Hyundai  2014  30000      Delhi       27
2      Tata  2011  60000     Mumbai       25
3  Mahindra  2015  25000      Delhi       26
4    Maruti  2012  10000     Mumbai       28
5   Hyundai  2016  46000      Delhi       29
6   Renault  2014  31000     Mumbai       24
7      Tata  2018  15000    Chennai       21
8    Maruti  2019  12000  Ghaziabad       24
```

Now using loc( ) let's select rows where cars of 'Hyundai' brand has been driven more than 40000 kms.

```
data = (df.loc[(df.Brand == 'Hyundai') & (df.Kms > 40000)])
print(data)
```

O/P:

```
     Brand  Year    Kms   City  Mileage
5  Hyundai  2016  46000  Delhi       29
```

Now let's use the row range selector syntax with loc( ) to select specific rows from the dataframe.

```
data = (df.loc[2 : 4])
Print(data)
```

O/P:

```
      Brand  Year    Kms    City  Mileage
2      Tata  2011  60000  Mumbai       25
3  Mahindra  2015  25000   Delhi       26
4    Maruti  2012  10000  Mumbai       28
```

Difference between loc( ) and iloc( )

- loc() is label based data selecting method which means that we have to pass the name of the row or column which we want to select. iloc( ) is integer index based data

selecting method, which means that it only takes integer index as arguments for data selection.

- The loc( ) indexer can also do boolean selection. For instance, if we are interested in finding all the rows where Age is less 30 and return just the Color and Height columns we can do that sing loc( ). On the other hand such operation is not that simple to do with iloc( ), though possible but let's just say that iloc( ) is not suitable for Boolean selection.

Using iloc( ) for data selection

Pandas provide a unique method to retrieve rows from a Data frame. Dataframe.iloc[] method is used when the index label of a data frame is something other than numeric series of 0, 1, 2, 3….n or in case the user doesn't know the index label.

Syntax: pandas.DataFrame.iloc[]

Parameter: Index position of rows in integer or list of integer.

```
import pandas
students = [('Amar', 17, 'FYBCA', 75),
           ('Akbar', 18, 'FYBCA', 86),
           ('Anthoni', 18, 'SYBCA ', 67),
           ('Joli', 19, 'SYBCA', 81),
           ('Viru', 19, 'TYBCA', 77),
           ('Jay', 20, 'TYBCA', 83)]
df=
pandas.DataFrame(students,columns=['Name','Age','Class','Percent'])
print(df)
data = df.iloc[3]
print(data)
print("---------------------------------")
data = df.iloc[[1,3,5]]
print(data)
print("---------------------------------")
data = df.iloc[2:4]
print(data)
```

O/P:

```
Name        Joli
Age           19
Class      SYBCA
Percent       81
Name: 3, dtype: object
---------------------------------
    Name  Age  Class  Percent
1  Akbar   18  FYBCA       86
3   Joli   19  SYBCA       81
5    Jay   20  TYBCA       83
---------------------------------
      Name  Age   Class  Percent
2  Anthoni   18  SYBCA        67
3     Joli   19   SYBCA       81
```