Introduction to Python programming

Python is a powerful high-level, object-oriented programming language. It has simple easy-to-use syntax. Python is a general-purpose language. It has wide range of applications from Web development(Django and Flask), scientific and mathematical computing (SciPy, NumPy)

Feature of Python

1. A simple language which is easier to learn
   Python has a very simple and elegant syntax. It's much easier to read and write Python programs compared to other languages like: C++, Java, C#. Python makes programming fun and allows you to focus on the solution rather than syntax.
2. Free and open-source
   Python can be distributed freely, even for commercial use. Anyone can make changes to the Python's source code. Python has a large community constantly improving it in each iteration.
3. Portability
   You can move Python programs from one platform to another, and run it without any changes. It runs seamlessly on almost all platforms including Windows, Mac OS X and Linux.
4. Extensible and Embeddable
   Python code can be integrated easily with other languages like C/C++, R etc. This gives the application high performance as well as scripting capabilities which other languages may not provide.
5. A high-level, interpreted language
   Unlike C/C++, user don't have to worry about memory management, garbage collection and so on.
6. Large standard libraries to solve common tasks
   Python has a number of standard libraries which makes life of a programmer much easier since user don't have to write all the code. For example: if the application Need to connect MySQL database on a Web server, it is done using MySQLdb
7. Python is both functional and Object Oriented Programming Language

How Python is different from Other Programming Languages

- Simple syntax
- Python code is 5 to 10 times shorter than equivalent C++ code and 3 to 5 times shorter than equivalent Java code
- Python function can return multiple values
- It supports higher order functions. functions within a function and passing function as a arguments

Example of Code Simplicity

Below is the example code in Java C++ and Python

Java Code

```
public class example {
    public static void main(String[] args)
        {
            System.out.println("Hello Duratech");
        }
    }
```

C++ Code

```
#include <iostream>
void main()
{
cout << "Hello Duratech";
}
```

Python Code

```
print("Hello Duratech")
```

Example 2

Consider the case if a there are set of numbers from 1 to 10. Multiplying all the elements with 2. The same code is written in Java and Python

Java Code

```
import java.util.ArrayList;
import java.util.Arrays;

public class test {
   public static void main(String args[])
    {
        List<Integer> numbers = Arrays.asList(1, 2, 3,4, 5, 6, 7, 8);
        List<Integer> result = null;
        for (int i = 0; i < numbers.size(); i++) {
            result.add(numbers.getIndex(i)*2)
        }
    }
}
```

Python Code

```
import numpy as np
numbers = np.array([1, 2, 3, 4, 5, 6,7, 8,9,10])
numbers*2
```

List Comprehension

List comprehensions is better way to create lists based on existing lists. When using list comprehensions, lists can be built by leveraging any iterable items.

Below code consist of a creating a list and iterating those items and also filtering

```
number_list = [x for x in range(100) if x % 3 == 0 if x % 5 == 0]
print(number_list)
```

Python program files

- Python code is usually stored with extension " `.py` ":

```
myprogram.py
```

- To run Python program from the command line:

```
$ python myprogram.py
```

Variables and types

Symbol names

Variable names in Python can contain alphanumerical characters `a-z` , `A-Z` , `0-9` and some special characters such as `_` . Variable names must start with a letter.

By convention, variable names start with a lower-case letter, and Class names start with a capital letter.

In addition, there are a number of Python keywords that cannot be used as variable names. These keywords are:

```
and, as, assert, break, class, continue, def, del, elif, else, except,
exec, finally, for, from, global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while, with, yield
```

Assignment

The assignment operator in Python is `=` . Python is a dynamically typed language, so the user do not need to specify the type of a variable.

Assigning a value to a new variable creates the variable:

In [1]:

```
# variable assignments
x = 1.0
```

Fundamental types

Python has various data types

- Integer

  ```
  # Example for an  integer
  x=1
  ```

- Float

  ```
  # Example for a float
  x=1.5
  ```

- String

  ```
  # Example for a string
  x= "Duratech"
  ```

- Boolean

  ```
  # Example for  a boolean
  x= True
  ```

- Complex

  ```
  # Example for a complex
  x= 3 + 4j
  ```

In [2]:

```
# integers
x = 1
type(x)
```

Out[2]:

```
int
```

In [3]:

```
# float
x = 1.0
type(x)
```

Out[3]:

```
float
```

In [4]:

```
# boolean
b1 = True
b2 = False

type(b1)
```

Out[4]:

```
bool
```

In [5]:

```
# complex numbers: note the use of `j` to specify the imaginary part
x = 1.0 - 1.0j
type(x)
```

Out[5]:

```
complex
```

In [6]:

```
print(x)
```

```
(1-1j)
```

In [7]:

```
print(x.real, x.imag)
```

```
1.0 -1.0
```

If variable is used that has not yet been defined `NameError` occurs:

In [8]:

```
print(y)
```

```
--------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-8-d9183e048de3> in <module>
----> 1 print(y)

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

Type utility functions

The module `type` contains a number of type name definitions that can be used to test if variables are of certain types:

In [1]:

```python
x = 1.0

# check if the variable x is a float
type(x) is float
```

Out[1]:

```
True
```

In [2]:

```python
# check if the variable x is an int
type(x) is int
```

Out[2]:

```
False
```

`isinstance` method can be also used for testing types of variables:

In [3]:

```python
isinstance(x, float)
```

Out[3]:

```
True
```

Type casting

In [4]:

```python
x = 1.5

print(x, type(x))
```

```
1.5 <class 'float'>
```

Some type casting

Function	Description
int(x)	convert x to an integer number
str(x)	convert x to a string
chr(x)	convert x to a string
float(x)	convert x to a floating point number
hex(x)	convert x to a hexadecimal string
oct(x)	convert x to a an octal string

Operators and comparisons

Most of the operators and comparisons work well with Python

Arithmetic Operators

operators	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus or Remainder
**	Power

In [5]:

```
# Addition
1 + 2
```

Out[5]:

```
3
```

In [6]:

```
# Subtraction
20 - 10
```

Out[6]:

```
10
```

In [7]:

```
# Multiplication
30 * 10
```

Out[7]:

```
300
```

In [8]:

```
# Division
1000/125
```

Out[8]:

```
8.0
```

In [9]:

```
# Division by float. remainder will be an float
45/2
```

Out[9]:

```
22.5
```

In [10]:

```
# Division by integer.This always returns a integer value
47//2
```

Out[10]:

```
23
```

In [11]:

```
# Modulus . Gives the remiander
5%4
```

Out[11]:

```
1
```

In [12]:

```
# Power operator
9**3
```

Out[12]:

```
729
```

Logical Operator

Operators	Description
and	Logical And
or	Logical Or
not	Logical Not

In [13]:

```
True and False
```

Out[13]:

```
False
```

In [14]:

```
not False
```

Out[14]:

```
True
```

In [15]:

```
True or False
```

Out[15]:

```
True
```

Relational Operator

Operators	Description
>	Greater than
<	Lesser than
>=	Greater than and equal to
<=	Lesser than and equal to
==	Equality
!=	Not Equal

In [16]:

```
a=10
b=5
c=5
a < b, b <= c, a > b, a >= b, b==c, b!=c
```

Out[16]:

```
(False, True, True, True, True, False)
```

In [17]:

```
# equality
[1,2] == [1,2]
```

Out[17]:

```
True
```

In [18]:

```
# objects identical?
l1 = l2 = [1,2]
l1 is l2
```

Out[18]:

```
True
```

Bitwise Operator

In [ ]:

```
a = 20             # 20 = 0001 0100
b = 13            # 13 =  0000 1101
```

In [19]:

```
#Bitwise And
a&b
```

Out[19]:

```
0
```

In [20]:

```
#Bitwise or
a | b
```

Out[20]:

```
15
```

In [21]:

```
# Bitwise Xor

a^b
```

Out[21]:

```
15
```

In [22]:

```
# Complimentary
~a
```

Out[22]:

```
-11
```

Input and Output Operator

input()

This function first takes the input from the user and then evaluates the expression

In [19]:

```
name=input("Enter the Name:")
print(name)
```

```
Enter the Name:Duratech
Duratech
```

The value will be always a string. If needed it has to be type casted

In [23]:

```
age= input("Enter the age: ")
type(age)
```

```
Enter the age: 12
```

Out[23]:

```
str
```

In [24]:

```
age = int(input("Enter the age: "))
type(age)
```

```
Enter the age: 23
```

Out[24]:

```
int
```

In [30]:

```
##### Getting multiple values
m, n = input("Enter a two value: ").split()
print(m)
print(n)
```

```
Enter a two value: 43 65
43
65
```

Getting list of values

Python has ability to give multiple inputs with space as seperator and can store it as list. List will be dealt later it this document,

In [32]:

```
### Getting list of values
x = list(map(int, input("Enter values: ").split()))
x
```

```
Enter values: 43 65 76 87 32
```

Out[32]:

```
[43, 65, 76, 87, 32]
```

Output

The output is being performed by print statement

```
print()
```

It can print any values and variables

In [36]:

```
x= "Python"
print("Duratech")
print(10)
print(x)
```

```
Duratech
10
Python
```

In [42]:

```
# printint Multiple times

print("Nadal "*3)
```

```
Nadal Nadal Nadal
```

In [46]:

```
# Print multiple values

print(10,20,30)

# It can print all kinds of values
print("Cricket ",2019, True)


print('Football', 'Worldcup', 2022, sep=',')
```

```
10 20 30
Cricket  2019 True
Football,Worldcup,2022
```

In [68]:

```
country="Qatar"
print("Next world cup football takes place in ",country)
```

```
Next world cup football takes place in  Qatar
```

In [69]:

```
## Print with parameters
print ("USA has {} states ".format(50))    # With single input

print ("India has {} states and {} Union Territories".format(29,9))  # with 2 in
puts
```

```
USA has 50 states
India has 29 states and 9 Union Territories
```

In [67]:

```
# Formatting the strings
print ("My average marks was {0:1.2f}%".format(78.234876))
```

```
My average marks was 78.23%
```

Control Flow

Conditional statements: if, elif, else

The Python syntax for conditional execution of code uses the keywords `if` , `elif` (else if), `else` :

Indentation

In Python, indentation is used to mark a block of code. In order to indicate a block of code, there should indent each line of the block of code by four spaces

```
if condition1:
    if condition2:
        statement
    else
        stataments
```

If Statement

`if statement` executes statement only if a specified condition is true.

```
if condition:
    statements
```

In [23]:

```
a = 200
b = 100
if a > b:
    print ("a is greater than b")
```

```
a is greater than b
```

If else Statement

The statement under `if` executes statement only if a specified condition is true otherwise else part is executed.

```
if condition:
    statement 1
else:
    statement 2
```

In [24]:

```
a = 100
b = 200
if a > b:
    print ("a is greater")
else:
    print (" b is greater")
```

```
 b is greater
```

If-elif-else Statement

If there are multiple statement to be met Then `if elif  else` statement should be used

```
if condition1:
    statement1

elif condition2:
    statement 2

else:
    default statement
```

In [25]:

```
num=200
if num >0:
    print ("Positive Number")
elif num<0:
    print ("Negative Number")
else:
    print ("Zero")
```

```
Positive Number
```

Loops

In Python, loops can be programmed in a number of different ways. There few kinds of loops used here like `for` and `while`

`for` loops:

The most commonly used loop is the `for` loop, which is used together with iteratable objects, such as lists. The basic syntax is:

```
for val in sequence:
    Statements
```

Here `val` is the variable that takes the value of the item inside the sequence on each iteration.

In [26]:

```
for x in [1,2,3]:
    print(x)
```

```
1
2
3
```

In [27]:

```
for word in ["Cat","Dog","Elephant"]:
    print(word)
```

```
Cat
Dog
Elephant
```

In [28]:

```
# For loop for set
num_set = set([0, 1, 2, 3, 4, 5])
for n in num_set:
    print(n)
```

```
0
1
2
3
4
5
```

while loops:

These are loops that runs till a condition is met

```
while condition
    statements
```

In [29]:

```
i = 1
while i <= 5:
    print(i)
    i = i + 1
```

```
1
2
3
4
5
```

Continue

`continue` can skip the next command and continue the next iteration of the loop.

In the below output that the output has no 3.

```
if a==3:
        continue

    skips the next command continue the next while loop
```

In [30]:

```
a=0
while a<5:
    a = a + 1
    if a==3:
        continue
    print(a)
```

```
1
2
4
5
```

Break Statement

`break` keyword is used to stop the running of a loop according to the condition.

When the `num` reaches 5 it breaks the loop and execution stops

In [31]:

```
num=0
while num<10:
    if num==5:
        break
    num=num+1
    print(num)
```

```
1
2
3
4
5
```

Data Structure

It is a Collection of related data. It is a way of organizing and storing data so that it can be accessed efficiently. There are four types of Data structures in python.

- List
- Tuples
- Dictionary
- Set
- Frozen set

List

Lists are ordered collection of items of any data type.It is mutable which means that it can be modified any time.

The syntax for creating lists in Python is `[...]` :

```
listName = [val1, val2, val3]
```

In [32]:

```
# list can take multiple values it can be combination of string, numbers, boolean etc.,
s2 = [12,"red",False]
s2
```

Out[32]:

```
[12, 'red', False]
```

In [33]:

```
# in python 3 version range generates an iterator, which can be converted to a list using 'list(...)'.
start = 1
stop = 50
step =3
list(range(start, stop, step))
```

Out[33]:

```
[1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49]
```

In [34]:

```
# convert a string to a list by type casting:
s3 = "Hello world"
print(s3)
s4 = list(s3)

s4
```

```
Hello world
```

Out[34]:

```
['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

List Functions

Function	Description
list.append(n)	add an item to the end of the list
list.insert(i,n)	Insert an item at the specified index i
list.pop(i)	Remove & return the item at index i
list.remove(n)	removes the value n
list.reverse()	Reverse the list
list.index(n)	Return the index of given value
list.sort()	Sort the element of list increasingly
list.extend(lst)	Append each item of lst to list
list.clear	remove all items from the list

In [35]:

```
# create a new empty list
a1= [10,20,30,40,50]

# add an elements using `append`
a1.append(60)
a1.append(70)

print(a1)
```

```
[10, 20, 30, 40, 50, 60, 70]
```

In [36]:

```
a1.insert(1,15)
print(a1)
```

```
[10, 15, 20, 30, 40, 50, 60, 70]
```

In [37]:

```
s=a1.pop(2)
print(s,a1)
```

```
20 [10, 15, 30, 40, 50, 60, 70]
```

In [38]:

```
a1.remove(60)
a1
```

Out[38]:

```
[10, 15, 30, 40, 50, 70]
```

In [39]:

```
a1.reverse()
a1
c=a1.reverse()
```

In [40]:

```
a1.index(10)
```

Out[40]:

```
0
```

In [41]:

```
a1.sort()
a1
```

Out[41]:

```
[10, 15, 30, 40, 50, 70]
```

Difference between append and extend

Append add the element to the list at the end.

Extends iterates over its argument adding each element to the list, extending the list.

In [42]:

```
x = [1, 2, 3]
x.append([4, 5])
print (x)
```

```
[1, 2, 3, [4, 5]]
```

In [43]:

```
x = [1, 2, 3]
x.extend([4, 5])
print (x)
```

```
[1, 2, 3, 4, 5]
```

Replacing the element

In [44]:

```
# Replacing a single element
l = [10,20,50,40,50,80,70]
l[4]=100
l
```

Out[44]:

```
[10, 20, 50, 40, 100, 80, 70]
```

In [45]:

```
### Replacing multiple  elements
l = [10,20,50,40,50,80,70]
l[1:3]=[100,300]
l
```

Out[45]:

```
[10, 100, 300, 40, 50, 80, 70]
```

Remove an element at a specific location using `del` :

In [46]:

```
l = [ 10,20,30,40,50,60,70,80,90,100,110]
print(l)
del l[7]   # removes 80
del l[6]   # removes 70

print(l)
```

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110]
[10, 20, 30, 40, 50, 60, 90, 100, 110]
```

In [47]:

```
# Adding an list
lst1 = [0, 1, 2]
lst2 = [3, 4, 5]
list3 = lst1 + lst2
list3
```

Out[47]:

```
[0, 1, 2, 3, 4, 5]
```

In Operator

This returns TRUE if a element is present in list. This can be applied to numreic string etc.

In [9]:

```
15 in [1,2,8,10,15,30]
```

Out[9]:

```
True
```

In [10]:

```
"Sachin" in ["Rahul", "Virat","Dhoni"]
```

Out[10]:

```
False
```

In [11]:

```
"Sachin" not in ["Rahul", "Virat","Dhoni"]
```

Out[11]:

```
True
```

Tuples

Tuples are like lists, except that they cannot be modified once created, that is they are *immutable*.

In Python, tuples are created using the syntax `(..., ..., ...)` , or even `..., ...` :

```
tupleName = (val1, val2, val3)
tupleName = val1, val2, val3
```

In [12]:

```
point = (10, 20)

print(point, type(point))
```

```
(10, 20) <class 'tuple'>
```

In [13]:

```
point1 = 100, 200

print(point1, type(point1))
```

```
(100, 200) <class 'tuple'>
```

We can unpack a tuple by assigning it to a comma-separated list of variables:

Tuple Function

Function	Description
x in tpl	return true if x is in the tuple
len(tpl)	return length of tuple
tpl.count(x)	count how many x in tuple
tpl.index(x)	return the index of x

In [14]:

```
# Returns true if the element in present else returns false
tpl = (10,20,30,40,50)
x= 10
x in tpl
```

Out[14]:

```
True
```

In [15]:

```
# Returns true if the element in present else returns false
tpl = (10,20,30,40,50)
x= 70
x in tpl
```

Out[15]:

```
False
```

In [16]:

```
# Returns len of the tuple
tpl = (10,20,30,40,50)
len(tpl)
```

Out[16]:

```
5
```

In [17]:

```
# Returns count of particular element in a tuple
tpl = (10,20,30,40,50,10,10)
tpl.count(10)
```

Out[17]:

```
3
```

In [18]:

```
# Returns index value of element
tpl = (10,20,30,40,50)
tpl.index(30)
```

Out[18]:

```
2
```

In [19]:

```
# Adding Two tuples

tup1 = (10,20,30,40)
tup2 = ("Red","Blue")
tup3 = (True,False)

tup4 = tup1 + tup2 + tup3;
tup4
```

Out[19]:

```
(10, 20, 30, 40, 'Red', 'Blue', True, False)
```

In [20]:

```
# Creating a tuple

tup = ('physics', 'chemistry', 1997, 2000);
tup
```

Out[20]:

```
('physics', 'chemistry', 1997, 2000)
```

In [21]:

```
# Deleting the entire tuple
del tup
```

In [22]:

```
tup
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-22-95b80b2375ef> in <module>
----> 1 tup

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

Set

A set is an unordered collection with no duplicate elements. It removes duplicate entries and performs set operations such as intersection, union and difference. The set type is mutable.

Set can be created in two different ways either a list or using a curly bracket { }

```
# e.g.
x = {}
```

- set removes the duplicate.
- It is same as binary search. So search is faster in sets.

In [70]:

```
#Example
my_set = set([1,2,3,2])
my_set
```

Out[70]:

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

In [71]:

```
type(my_set)
```

Out[71]:

```
set
```

In [72]:

```
A = {1, 2, 3, 4, 5}
A
```

Out[72]:

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

In [73]:

```
type(A)
```

Out[73]:

```
set
```

set Function

Function	Description
set.copy()	copy the set
set.update(a, b, c)	Add a, b, c to the set
set.remove(n)	Remove the item n
set.pop()	Remove one random item
set1.intersection(set2)	Return items in both sets
set1.difference(set2)	Return items in set1 not in set2

In [74]:

```
# Copying  a set

my_set = set([1,2,3,2])
new_set = my_set.copy()
new_set
```

Out[74]:

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

In [75]:

```
# Adding an element
my_set = set([1,2,3,2])
print(my_set)
my_set.add(100)
my_set
```

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

Out[75]:

```
{1, 2, 3, 100}
```

In [76]:

```
# Updating a set. it is aimilar to concantenating two sets
A = set([1,2,3,2])
print(A)

B = {100,200,300}

A.update(B)
A
```

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

Out[76]:

```
{1, 2, 3, 100, 200, 300}
```

In [77]:

```
# Discarding a set
my_set = set([1,2,3,2])
print(my_set)

# discrading an element
my_set.discard(2)
my_set
```

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

Out[77]:

```
{1, 3}
```

In [78]:

```
# Remove an element
A = set([1,2,3,2])
A.remove(3)
A
```

Out[78]:

```
{1, 2}
```

In [79]:

```
my_set = set("HelloWorld")
print(my_set)

# pop an element
# Output: random element
print(my_set.pop())
```

```
{'e', 'H', 'o', 'W', 'l', 'r', 'd'}
e
```

Python Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference.

In [80]:

```
# Add

A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
#union
print(A | B)

# Union

print(A.union(B))
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
{1, 2, 3, 4, 5, 6, 7, 8}
```

In [81]:

```
# Intersection

A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

A.intersection(B)
```

Out[81]:

```
{4, 5}
```

In [82]:

```
# use difference function on A
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
A.difference(B)
```

Out[82]:

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

In [83]:

```
# use symmetric_difference function on A Removes the common elements from A and
 B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
A.symmetric_difference(B)
```

Out[83]:

```
{1, 2, 3, 6, 7, 8}
```

FrozenSet

Frozen set is an immutable version of a Python set object. Elements cannot be modifed in frozen sets an it can be modified in sets

In [84]:

```
vowels = ('a', 'e', 'i', 'o', 'u','a','o')

fSet = frozenset(vowels)
fSet
```

Out[84]:

```
frozenset({'a', 'e', 'i', 'o', 'u'})
```

Dictionary

A dictionary is a collection of key:value pair which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets.

The syntax :

```
dictionaryName = {key1:val1, key2:val2,...}
```

In [85]:

```
sampledict ={
  "brand": "Ford",
  "model": "Fiesta",
  "year": 2005
}

sampledict
```

Out[85]:

```
{'brand': 'Ford', 'model': 'Fiesta', 'year': 2005}
```

In [86]:

```
# by giving key name, value can be retrieved:
sampledict.get("model")
```

Out[86]:

```
'Fiesta'
```

In [87]:

```
# iterating the dictionary
squares = {1: 1, 3: 9, 5: 25, 7: 49}
for i in squares:
    print(squares[i])
```

```
1
9
25
49
```

Dictionary Function

Function	Description
d.items()	returns the key:value pairs of d
d.keys()	return the keys of d
d.values()	returns values of d
d.get(key)	return the values with specified key
d.pop(key)	removes and returns the value
d.clear()	Clears the dictionary
d.copy()	copy all items of d
d1.update(d2)	add key:value in d1 to d2

In [88]:

```
# create a dictionary
student = {"Name":"George", "RollNo":1234, "Age":20, "Marks":50}
student
```

Out[88]:

```
{'Name': 'George', 'RollNo': 1234, 'Age': 20, 'Marks': 50}
```

In [89]:

```
# Returns key value pair
student = {"Name":"George", "RollNo":1234, "Age":20, "Marks":50}
student.items()
```

Out[89]:

```
dict_items([('Name', 'George'), ('RollNo', 1234), ('Age', 20), ('Mar
ks', 50)])
```

In [90]:

```
# Returns all the keys
student = {"Name":"George", "RollNo":1234, "Age":20, "Marks":50}
student.keys()
```

Out[90]:

```
dict_keys(['Name', 'RollNo', 'Age', 'Marks'])
```

In [91]:

```
# Returns all the values
student = {"Name":"George", "RollNo":1234, "Age":20, "Marks":50}
student.values()
```

Out[91]:

```
dict_values(['George', 1234, 20, 50])
```

In [92]:

```
# Returns the value based on the key
student = {"Name":"George", "RollNo":1234, "Age":20, "Marks":50}
print(student.pop("Name"))
student
```

```
George
```

Out[92]:

```
{'RollNo': 1234, 'Age': 20, 'Marks': 50}
```

In [93]:

```
# Returns the value based on the key
student = {"Name":"George", "RollNo":1234, "Age":20, "Marks":50}
student.get("Age")
```

Out[93]:

```
20
```

In [94]:

```
# Adding new values

student = {"Name":"George", "RollNo":1234, "Age":20, "Marks":50}
student.update({"Rank":10})
student
```

Out[94]:

```
{'Name': 'George', 'RollNo': 1234, 'Age': 20, 'Marks': 50, 'Rank': 1
0}
```

In [95]:

```
# copying an data dictionary

x= student.copy()
x
```

Out[95]:

```
{'Name': 'George', 'RollNo': 1234, 'Age': 20, 'Marks': 50, 'Rank': 1
0}
```

In [96]:

```
# delete a particular item with a key
student = {"Name":"George", "RollNo":1234, "Age":20, "Marks":50}
del student["Marks"]
student
```

Out[96]:

```
{'Name': 'George', 'RollNo': 1234, 'Age': 20}
```

In [97]:

```
# remove all items
student = {"Name":"George", "RollNo":1234, "Age":20, "Marks":50}
student.clear()
student
```

Out[97]:

```
{}
```

Overview of Data Dictionary

Data Structure	Description	Syntax
List	store multiple changeable values	[ ]
Dictionary	store multiple key:value pairs	{ key,value }
Set	store multiple unique values	{ }
Tuple	store multiple unchangeable values	( )
Frozen set	store multiple and immutable unique values	frozenset()

Functions

Python has two sets of functions

- Built in functions
- User Defined functions

Built in functions

The Python has a number of functions built. e.g. `abs` `sum` There are lots of packages available. Each package has a collection of functions. Predefined functions can be imported using import function.

In [98]:

```
import math

math.sqrt(64)
```

Out[98]:

```
8.0
```

Math Function

Python has many built in function; one of the most useful modules is Math module.

Some Common Function

Function	Description
abs(n)	absolute value of n
round(n)	round off value to n
ceil(n)	returns an integer that is greater than or equal to its argument
floor(n)	returns an integer that is less than or equal to its argument
max(n, m)	returns the maximum value between two numbers
min(n,m)	returns the minimum value between two numbers
degree(n)	convert radians to degrees
radians(n)	convert degrees to radians
sqrt(n)	square root value of n
cos(n)	cosine value of n
sin(n)	sine value of n
log(n)	logarithm value of n
exp(n)	exponential value of n
pow(m,n)	Gives value of m^n

In [99]:

```
import math
abs(-10)
```

Out[99]:

```
10
```

In [100]:

```
max(10,40,60,43)
```

Out[100]:

```
60
```

In [101]:

```
min(10,40,43,60)
```

Out[101]:

```
10
```

In [102]:

```
pow(4,2)
```

Out[102]:

```
16
```

In [103]:

```
math.ceil(10.3)
```

Out[103]:

```
11
```

In [104]:

```
math.floor(10.4)
```

Out[104]:

```
10
```

In [105]:

```
math.degrees(math.pi)
```

Out[105]:

```
180.0
```

In [106]:

```
math.radians(180)
```

Out[106]:

```
3.141592653589793
```

In [107]:

```
round(100.4356,2)
```

Out[107]:

```
100.44
```

In [108]:

```
math.log(5)
```

Out[108]:

```
1.6094379124341003
```

In [109]:

```
math.exp(1.609)
```

Out[109]:

```
4.997810917177775
```

String Functions

Strings are the variable type that is used for storing text messages. Python has lots of functions related to strings

In [110]:

```
s = "Hello world"
type(s)
```

Out[110]:

```
str
```

In [111]:

```
# length of the string: the number of characters
len(s)
```

Out[111]:

```
11
```

In [112]:

```
# replace a substring in a string with something else
s2 = s.replace("world", "test")
print(s2)
```

```
Hello test
```

String formatting examples

In [113]:

```
print("str1", "str2", "str3")  # The print statement concatenates strings with a space
```

```
str1 str2 str3
```

In [114]:

```
# this formatting creates a string
s2 = "value1 = %.2f. value2 = %d" % (3.1415, 1.5)

print(s2)
```

```
value1 = 3.14. value2 = 1
```

Testing Function

Function	Description
isdigit()	return true if all characters are numbers
isalpha()	return true if all characters are alphabets
isupper()	return true if all characters are of Upper case
islower()	return true if all characters are of lower case
istitle()	return true if all characters are of title case
isspace()	return true if all characters are space

In [115]:

```
s4 = "1234"
s4.isdigit()
```

Out[115]:

```
True
```

In [116]:

```
s4 = "Sachin"
s4.istitle()
```

Out[116]:

```
True
```

In [117]:

```
s5="Bombay"
s5.isalpha()
```

Out[117]:

```
True
```

In [118]:

```
s6="DELHI"
s6.isupper()
```

Out[118]:

```
True
```

In [119]:

```
s7=" "
s7.isspace()
```

Out[119]:

```
True
```

Search Function

Function	Description
find(x)	return the index of first occurrence, or -1
rfind(x)	return the index of first occurence from right or -1
index(x)	return the index of first occurrence, or alert error
rindex(x)	return the index of first occurrence from right, or alert error

In [120]:

```
# Returns the index of first occurance
s1 = "Duratech Solutions"
s1.find("t")
```

Out[120]:

```
4
```

In [121]:

```
# Searches from right and return the index
s2 = "Duratech Solutions"
s2.rfind("t")
```

Out[121]:

```
13
```

In [122]:

```
s3 = "abec"
s3.index("e")
```

Out[122]:

```
2
```

Split Function

Function	Description
split(separator)	split a string by a separator
partition(separator)	partition a string by a separator to three parts

In [123]:

```
str = "Python is a very good language"
str.split(" ")
```

Out[123]:

```
['Python', 'is', 'a', 'very', 'good', 'language']
```

In [124]:

```
# separates the email to three parts. (head, separator, trail)
email = "test@abc.com"
email.partition(".")
```

Out[124]:

```
('test@abc', '.', 'com')
```

Other Functions

In [125]:

```
# joins the string with sepeartor
strDate = "-".join(["10","11","2018"])
strDate
```

Out[125]:

```
'10-11-2018'
```

In [126]:

```
# Swaps the case
str = "United States"
str.swapcase()
```

Out[126]:

```
'uNITED sTATES'
```

In [127]:

```
# Fills the string with Zero
str = "12346"
str.zfill(10)
```

Out[127]:

```
'0000012346'
```

Regular Expressions

Regular Expressions are used to match the string with specified pattern, performs the tasks of search, replacement and splitting.

Python has a built-in package called re, which can be used to work with Regular Expressions.

Few Regex Functions

Function	Description
findAll	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

In [128]:

```
import re
# Search for all values of la
txt = "Cricket has 11 players, they play for 5 days "
re.findall("la", txt)
```

Out[128]:

```
['la', 'la']
```

In [129]:

```
#The search() function searches the string for a match
import re
re.search('dog', 'dog cat dog')
```

Out[129]:

```
<_sre.SRE_Match object; span=(0, 3), match='dog'>
```

In [130]:

```
# Match Command
import re
pattern = re.compile("^(\d{2})-(\d{2})-(\d{4})$")
valid = pattern.match("01-01-2000")
if valid:
    print ("Valid date!")
else:
    print("Invalid Date!")
```

```
Valid date!
```

In [131]:

```
#Sub command. To remove # from the list
import re
str = "1800 3000 9009 # It is the toll free of amazon india"
re.sub(r"#", "", str)
```

Out[131]:

```
'1800 3000 9009  It is the toll free of amazon india'
```

Regular Expression with Special Meaning

In [132]:

```
# Getting elements from a to g which is of lower case
import re
txt = "An Apple a day keeps doctor away"
re.findall("[a-g]", txt)
```

Out[132]:

```
['e', 'a', 'd', 'a', 'e', 'e', 'd', 'c', 'a', 'a']
```

Regex Expression Patterns

In [133]:

```
import re

txt = "Washington"

#Search for a sequence that starts with "Wa", followed by two (any) characters,
 and a "i":

re.findall("Wa..i", txt)
```

Out[133]:

```
['Washi']
```

In [134]:

```
# Find atleast one word
import re
txt = "India always have rain during month of June"
re.findall("June|July", txt)
```

Out[134]:

```
['June']
```

In [135]:

```
#The 'r' in front tells Python the expression is a raw string. In a raw string,
 escape sequences are not parsed.
# For example, '\n' is a single newline character. But, r'\n' would be two chara
cters: a backslash and an 'n'.
str = 'My email is  abc123@google.com, His email is  cde@gmail.com Her mail is s
ss@dasd.in'
emails = re.findall(r'[\w\.-]+@[\w\.-]+', str)
for email in emails:
    print(email)
```

```
abc123@google.com
cde@gmail.com
sss@dasd.in
```

User Defined Function

User defined functions in Python is defined using the keyword `def` , followed by a function name, a signature within parentheses `()` , and a colon `:` . Function body should have one additional level of indentation.

In [136]:

```
def testfunction():
    print("test")
```

In [137]:

```
testfunction()
```

```
test
```

Functions can return a value with the `return` keyword:

In [138]:

```
def square(x):
    """
    Return the square of x.
    """
    return x ** 2
```

In [139]:

```
square(4)
```

Out[139]:

```
16
```

In [140]:

```
def  fact(x):
    fact =1
    while(x>1):
        fact = fact * x
        x-=1
    return fact
```

In [141]:

```
fact(4)
```

Out[141]:

```
24
```

In [142]:

```
def add(x, y=10):
    return x+y
```

If we don't provide a value of the `y` argument when calling the the function `add` it defaults to the value(10) provided in the function definition:

In [143]:

```
# Here it takes 5 as x and y value is not provided then it takes 10 default value for y
add(5)
```

Out[143]:

```
15
```

In [144]:

```
# if both values are given then it takes value from parameters
add(100,200)
```

Out[144]:

```
300
```

Errors and Exceptions

In Python, there are two kinds of errors

- syntax errors
- exceptions.

Syntax Error

It is also known as parsing error. It is due to invalid syntax or invalid intendation

In [145]:

```
# Syntax error

c=[1,2,3,45])
```

```
  File "<ipython-input-145-19889fd23ff3>", line 3
    c=[1,2,3,45])
                ^
SyntaxError: invalid syntax
```

Indentation Error

In Python indentation is must. Here like other language there are no brackets. Instead of that it requires indentation is must. At least 4 spaces are required

In [146]:

```
# indentation Error
#This is due to wrong intendation. In Python indentation is must
a= 10
if a>0:
print("Positive")
else:
print("Negative")
```

```
  File "<ipython-input-146-812f16f6f363>", line 5
    print("Positive")
        ^
IndentationError: expected an indented block
```

Correct indentation

```
a= 10
if a>0:
    print("Positive")
else:
    print("Negative")
```

Exceptions

Even if a statement or expression is syntactically correct, there may be an error when an attempt is made to execute it. Errors detected during execution are called exceptions. some of the exceptions are given below

In [147]:

```
# Divisible by Zero
1/0
```

```
---------------------------------------------------------------------
-------
ZeroDivisionError                         Traceback (most recent cal
l last)
<ipython-input-147-71233faae7dc> in <module>
      1 # Divisible by Zero
----> 2 1/0

ZeroDivisionError: division by zero
```

In [49]:

```
# variable not found
v + 4
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-49-5f18184f0cc1> in <module>
      1 # variable not found
----> 2 v + 4

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

Handling Exceptions

Exceptions are handled using try block.If an error is encountered, a try block code execution is stopped and transferred down to the except block. There is a finally block. The code in the finally block will be executed regardless of whether an exception occurs.

In [50]:

```
# example

try:
    x= 1/0
except ZeroDivisionError:
    print("You can't divide by zero!")
```

```
You can't divide by zero!
```

Some of the common exception errors are:

`IOError`
If the file cannot be opened.

`ImportError`
If python cannot find the module

`ValueError`
Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value

`KeyboardInterrupt`
Raised when the user hits the interrupt key (normally Control-C or Delete)

`ZeroDivisionError` :
Raised when denominator is zero in division

`EOFError`
Raised when one of the built-in functions (input() or raw_input()) hits an end-of-file condition (EOF) without reading any data

In [148]:

```
# Example to handle an exception
def divide(x, y):
    result = "Invalid Input"
    try:
        result = x / y
    except ZeroDivisionError:
            result = "Cannot divide a number by zero"
    finally:
            return(result)
```

In [149]:

```
# when an input is string
divide(1,"2")
```

Out[149]:

```
'Invalid Input'
```

In [150]:

```
# when 0 is in denomintor
divide(1,0)
```

Out[150]:

```
'Cannot divide a number by zero'
```

In [151]:

```
#NO errors
divide(10,2)
```

Out[151]:

```
5.0
```

File Operations

Python has in-built functions to create and manipulate files

opening a file

`Open()`
The built-in Python function open() is used to open the file

If the file is in working directory
`data=open("data.txt")`

If the file is in another path
`data1=open("D:\\data1.txt")`

Good option to open a file

```
try:
    data=open("D:\\data.txt")
except IOError:
    print("File not found or path is incorrect")
finally:
    print("exit")
```

Access Modes

Access modes defines the way in which the file should be opened, It specifies from where to start reading or writing in the file

Mode	Function
r	Open a file in read only mode. Starts reading from beginning of file. This is the default mode
rb	Open a file for reading in binary format. Starts reading from beginning of file
r+	Open file for reading and writing. File pointer placed at beginning of the file.
w	Open file for writing only. File pointer placed at beginning of the file. Overwrites existing file or creates a new one if it does not exists.
wb	Same as w but opens in binary mode
w+	Same as w but also allows to read from file.
wb+	Same as wb but also allows to read from file.
a	Open a file for appending. Starts writing at the end of file. Creates a new file if file does not exist.
ab	Same as a but in binary format. Creates a new file if file does not exist.
a+	Same a a but also open for reading.
ab+	Same a ab but also open for reading.

Writing into a file

```
new_file=open("D:\\data.txt",mode="w",encoding="utf-8")
new_file.write("Content1\n")
new_file.write("content2 \n")
new_file.close()
```

Reads lines

```
data=open("D:\\data1.txt")
data.read(3)
```

closing the file

The file can be closed using the close command

```
data.close()
```

Reading the file line by line

```
data=open("D:\\data1.txt","r")
for line in data:
    print(line)
data.close()
```

Deleting a File

A file can be removed using os.remove() function

To avoid getting an error, If it good to check if the file exist before it is tried to delete it:

Example Check if file exist then delete it:

```
import os
if os.path.exists("file.txt"):
    os.remove("file.txt")
else:
    print("The file does not exist")
```

Python Objects and Class

Classes are the key features of object-oriented programming. A class is a structure for representing an object and the operations that can be performed on the object.

Object is a collection of data (variables) and methods (functions) that act on those data. The class is a blueprint for the object.

A class is defined almost like a function, but uses the `class` keyword, and the class definition usually contains a number of class method definitions (a function in a class).

```
class ClassName: # define a class
    classVariable = value # declare a class variable
    def __init__(self): # declare a constructor,   This is like this clas
s in java
    def method(self): # define a class method
```

init(self): It is a constructor for initialization. It is called when an object is created.
self : is a variable that refers to the current object.
def method(self) : ” define a class method with argument self

In [152]:

```
# Defining a Class in Python
class Sampleclass():
    "Sample class"
    "This is a set of properties of the class"
```

In [153]:

```
# Example
class Student():
    courses = ["English", "Mathematics", "Maths"]
    age = 15


    def ageIncrement(self):
        """This method increments the age of the instance."""
        self.age += 1
```

Creating an Object

Object is an instance of a class.

```
objectName = ClassName( args )
```

In [154]:

```
john = Student()
john.age
```

Out[154]:

15

In [155]:

```
john.ageIncrement()
john.age
```

Out[155]:

16

In [156]:

```
class Student():
    def __init__(self, courses, age, sex):
        self.courses = courses
        self.age = age
        self.sex = sex

    def ageIncrement(self):
        self.age += 1
```

In [157]:

```
# Example for calculating price per square feet
class Rectangle:
    def __init__(self, length, breadth, unit_cost=0):
        self.length = length
        self.breadth = breadth
        self.unit_cost = unit_cost

    def get_perimeter(self):
        return 2 * (self.length + self.breadth)

    def get_area(self):
        return self.length * self.breadth

    def calculate_cost(self):
        area = self.get_area()
        return area * self.unit_cost
```

To create a new instance of a class:

In [158]:

```
r = Rectangle(100, 120, 1000)
```

In [159]:

```
print("Area: %s" % (r.get_area()))
print("Cost: Rs.%s " %(r.calculate_cost()))
```

```
Area: 12000
Cost: Rs.12000000
```

Inheritance

Inheritance can be performed in Python. It has two classes

- base class
- derived class

Syntax

```
class BaseClass:
    statements
class DerivedClass(BaseClass):
    statements
```

In [160]:

```
# creating a Truck class
class TruckClass:

    def __init__(self, name, colour):
        self.__name = name
        self.__colour = colour

    def getColour(self):
        return self.__colour

    def setColour(self, colour):
        self.__colour = colour

    def getName(self):
        return self.__name
```

In [161]:

```
class Truck(TruckClass):

    def __init__(self, name, colour, model):
        super().__init__(name, colour)     # Calling the base class
        self.__model = model

    def getDescription(self):
        return "Name: " + self.getName() + " Model :" + self.__model + " Colour
 " + self.getColour()
```

In method `getDescrition()` method `getName()` , `getColour()` is not defined, since it is the derived from `TruckClass` they are accessible to child class through inheritance

In [162]:

```
v = Truck("Volvo", "black", "VNL")
print(v.getDescription())
print(v.getName())
```

```
Name: Volvo Model :VNL Colour black
Volvo
```

Polymorphism

Polymorphism is the ability to take various forms. If the program has more than one class it has the ability to perform different method for different object

In [163]:

```
class Car: # define a class
    def wheel(self): # define a wheel() method
        print ("4 wheelers")
class Truck: # define a class
    def wheel(self): # define a wheel() method
        print ("8 wheelers")
```

In [164]:

```
# Calling as car
c = Car()
c.wheel()
```

```
4 wheelers
```

In [165]:

```
# Calling as Truck
t = Truck()
t.wheel()
```

```
8 wheelers
```

Numpy

Introduction

The `numpy` package (module) is used in almost all numerical computation using Python. It is a package that provide high-performance vector, matrix and higher-dimensional data structures for Python. NumPy is an incredible library to perform mathematical and statistical operations. It works perfectly well for multi-dimensional arrays and matrices multiplication.

NumPy is memory efficiency, meaning it can handle the vast amount of data more accessible than any other library. Besides, NumPy is very convenient to work with, especially for matrix multiplication and reshaping

NumPy is the fundamental package for scientific computing with Python. It contains

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- useful linear algebra, Fourier transform, and random number capabilities
- can also be used as an efficient multi-dimensional container of generic data.

Before he numpy packge is used. It should be imported using import package

```
import numpy as np
```

Arrays

A numpy array is a grid of values, all of the same type. The shape of an array is a tuple of integers giving the size of the array along each dimension.

Numpy are faster than iterating through the loop. Loops are inefficient compared to numpy operations. Consider the following example

In [166]:

```
import numpy as np
a = np.array([1,2])
b = np.array([2,1])

dot = 0

for e,f in zip(a,b):
    dot += e*f

dot
```

Out[166]:

```
4
```

In [167]:

```
# This can be given as
np.sum(a*b)
```

Out[167]:

```
4
```

In [168]:

```
from numpy import *
# a vector: the argument to the array function is a Python list
v = array([1,2,3,4,5,6,7,8,9,10])
v
```

Out[168]:

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [169]:

```
# Creating a numpy array from a array
import numpy as np
npa = np.array(v)
npa
```

Out[169]:

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [170]:

```
# consider an example of adding adding 2 to each number above created numpy array.
#It iterates through each value in the numpy array

npa2=npa + 2
npa2
```

Out[170]:

```
array([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

In [171]:

```
# To find size of an Array
print(npa.size)
```

```
10
```

In [172]:

```
a = np.array([1, 2, 3])  # Create a 1 dimension array

# Getting the index 1
print(a[1])
```

```
2
```

In [173]:

```
# Example for creating multiple array
import numpy as np
a = np.array([[10, 20, 30], [40, 50, 60]])
a
```

Out[173]:

```
array([[10, 20, 30],
       [40, 50, 60]])
```

In [174]:

```
# tofind the shape of array i.e. number of rows and columns
a.shape
```

Out[174]:

```
(2, 3)
```

In [175]:

```
# printing specific indexes

print(a[1,2])
print(a[1,1])
```

```
60
50
```

Reshape and Flatten Data

Converting rows into columns and colums into rows. It can rearranged the rows and columns

In [176]:

```
import numpy as np
e= np.array([(1,2,3), (4,5,6),(7,8,9),(10,11,12)])
print(e)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

In [177]:

```
e.reshape(6,2)   # Creates an 6 x 2 array with the values
```

Out[177]:

```
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10],
       [11, 12]])
```

In [178]:

```
e.flatten()   # gives the one dimensional series values
```

Out[178]:

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

Ways to create numpy arrays

np.zeros and np.ones

It creates a matrix full of zeroes or ones. It is use to initialize the matrix with zeros or ones.

In [179]:

```
import numpy as np
a = np.zeros((2, 2))
a
```

Out[179]:

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

In [180]:

```
a = np.ones((2, 3))
print(a)
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
```

In [181]:

```
b = np.full((3,3), 8) # Create a constant array
b
```

Out[181]:

```
array([[8, 8, 8],
       [8, 8, 8],
       [8, 8, 8]])
```

In [182]:

```
d = np.eye(3)        # Create a 3x3 identity matrix
d
```

Out[182]:

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

In [183]:

```
import numpy as np
e = np.random.random((2,2))  # Create an array filled with random values
e
```

Out[183]:

```
array([[0.10842964, 0.86275331],
       [0.75740158, 0.04158337]])
```

Array indexing

In [184]:

```
import numpy as np
s = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(s)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

In [185]:

```
# First row
print (s[ :1])
```

```
[[1 2 3 4]]
```

In [186]:

```
print(s[1:])
```

```
[[ 5  6  7  8]
 [ 9 10 11 12]]
```

Index slicing

Index slicing is the technical name for the syntax `[lower:upper:step]` to extract part of an array:

In [187]:

```
data= np.array([(1,2,3), (4,5,6),(7,8,9),(10,11,12)])
data
```

Out[187]:

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

In [188]:

```
# This gives data of column no 1 of an array:
data[:, 1]
```

Out[188]:

```
array([ 2,  5,  8, 11])
```

In [189]:

```
data[:, 1:2]
```

Out[189]:

```
array([[ 2],
       [ 5],
       [ 8],
       [11]])
```

In [190]:

```
# Use slicing to get the data of  first 2 rows and columns 1 and 2;
b = data[:2, 1:3]
b
```

Out[190]:

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

Boolean array indexing

Boolean array indexing helps to pick out arbitrary elements of an array.

In [52]:

```
import numpy as np

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

boolean_idx = (a > 2)
boolean_idx
```

Out[52]:

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

In [192]:

```
# Filtering
print(a[boolean_idx])  # Prints "[3 4 5 6]"

# Above steps could be given as
print(a[a > 2])
```

```
[3 4 5 6]
[3 4 5 6]
```

In [54]:

```
from numpy.random import randn
arr = np.array([[1,2], [3, 4], [5, 6]])
print(arr)

#np.where(cond, trueval, falseval)
np.where(arr<4, 100, 50)
```

```
[[1 2]
 [3 4]
 [5 6]]
```

Out[54]:

```
array([[100, 100],
       [100,  50],
       [ 50,  50]])
```

Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays.

In [194]:

```
import numpy as np

x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

```
int64
float64
int64
```

Operation on Numpy

In [195]:

```
x = np.array([[1,2],[4,5]])
y = np.array([[3,4],[6,7]])
# Addition
np.add(x, y)
```

Out[195]:

```
array([[ 4,  6],
       [10, 12]])
```

In [196]:

```
# subtraction
np.subtract(x,y)
```

Out[196]:

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

In [197]:

```
# Multiplication
np.multiply(x,y)
```

Out[197]:

```
array([[ 3,  8],
       [24, 35]])
```

In [198]:

```
# Division
np.divide(x, y)
```

Out[198]:

```
array([[0.33333333, 0.5       ],
       [0.66666667, 0.71428571]])
```

In [199]:

```
# Inner product of vector
np.dot(x,y)
```

Out[199]:

```
array([[15, 18],
       [42, 51]])
```

In [200]:

```
# Gives the outer product of vector
np.outer(x,y)
```

Out[200]:

```
array([[ 3,  4,  6,  7],
       [ 6,  8, 12, 14],
       [12, 16, 24, 28],
       [15, 20, 30, 35]])
```

In [201]:

```
## Basic mathematic operation

#max min sum:
a= np.array([1,2,3])
print(a.min())
print(a.max())
print(a.sum())
```

```
1
3
6
```

In [202]:

```
#Square Root
a=np.array([(1,2,3),(3,4,5,)])
print(np.sqrt(a))
```

```
[[1.         1.41421356 1.73205081]
 [1.73205081 2.         2.23606798]]
```

In [203]:

```
# trignometric Operation
a=np.array([(1,2,3),(3,4,5,)])
np.sin(a)
```

Out[203]:

```
array([[ 0.84147098,  0.90929743,  0.14112001],
       [ 0.14112001, -0.7568025 , -0.95892427]])
```

In [204]:

```
# Logarithmic operation

a=np.array([(1,2,3),(3,4,5,)])
np.log(a)
```

Out[204]:

```
array([[0.        , 0.69314718, 1.09861229],
       [1.09861229, 1.38629436, 1.60943791]])
```

In [205]:

```
# Exponential operation

a=np.array([(1,2,3),(3,4,5,)])
np.exp(a)
```

Out[205]:

```
array([[  2.71828183,   7.3890561 ,  20.08553692],
       [ 20.08553692,  54.59815003, 148.4131591 ]])
```

In [206]:

```
# Mean median

data = [1,2,3,5,6,7,8]

print(np.mean(data))
print(np.median(data))
```

```
4.571428571428571
5.0
```

In [207]:

```
# Standard Deviation and variance

std=np.std(data)
print(std)
variance = np.var(data)
print(variance)
```

```
2.4411439272335804
5.959183673469389
```

In [208]:

```
# percentiles 90th percentile

np.quantile(data,0.9)
```

Out[208]:

```
7.4
```

In [209]:

```
# Product

np.prod(data)
```

Out[209]:

```
10080
```

In [210]:

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

```
[[1 2]
 [3 4]]
```

In [211]:

```
y = np.array([[5,6],[7,8]])
print(y)
```

```
[[5 6]
 [7 8]]
```

In [212]:

```
# Command to find unique values

names = np.array(['Sai', 'Jude', 'Bala', 'Arun', 'Balaji', 'Sai', 'Bala', 'Varu
n'])

np.unique(names)
```

Out[212]:

```
array(['Arun', 'Bala', 'Balaji', 'Jude', 'Sai', 'Varun'], dtype='<U
6')
```

Matrix Operation

Matrix operation can be performed using numpy. Matrix addition, subtraction multiplication, inverse, determinant etc....

In [213]:

```
a = [[1, 0], [0, 1]]
b = [[4, 1], [2, 2]]
print(a)
print(b)
```

```
[[1, 0], [0, 1]]
[[4, 1], [2, 2]]
```

In [214]:

```
# Matrix Addition
np.add(a,b)
```

Out[214]:

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

In [215]:

```
# Matrix Subtraction
np.subtract(a,b)
```

Out[215]:

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

matrix multiplication

c(i,j) = a(i,j) * b(i,j)

In [216]:

```
np.matmul(a, b)
```

Out[216]:

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

In [217]:

```
#Other matrix operation can be also perfomed
a = [[1, 0], [0, 1]]

#inverse of matrix
ainv = np.linalg.inv(a)
ainv
```

Out[217]:

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

In [218]:

```
#matrix deteriminant
np.linalg.det(a)
```

Out[218]:

```
1.0
```

In [219]:

```
#matrix diagonal
print("The matrix diagonal ",np.diag(a))

# to create a diagonal  matrix
np.diag([1,2])
```

```
The matrix diagonal  [1 1]
```

Out[219]:

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

In [220]:

```
# Return the sum along diagonals of the array.
import numpy as np
a  = [[1, 3], [6, 8]]
print(a)
np.trace(a)
```

```
[[1, 3], [6, 8]]
```

Out[220]:

```
9
```

In [221]:

```
# Function to find covariance
import numpy as np
#eigen values & eigen vectors
x = np.random.randn(100,3)
#to get covariance of matrix
cov = np.cov(x.T)
print(cov)
```

```
[[ 1.17985839 -0.16453279  0.0977972 ]
 [-0.16453279  0.94984228 -0.06597323]
 [ 0.0977972  -0.06597323  1.06762249]]
```

In [222]:

```
#Compute the eigenvalues and right eigenvectors of a square array.
np.linalg.eig(cov)
```

Out[222]:

```
(array([1.32009698, 1.01415927, 0.86306691]),
 array([[-0.79839301,  0.41054478,  0.44047882],
        [ 0.42990977, -0.1235395 ,  0.89438   ],
        [-0.42159957, -0.90343289,  0.07786414]]))
```

In [223]:

```
#Return the eigenvalues and eigenvectors of a Hermitian or symmetric matrix.
np.linalg.eigh(cov)
```

Out[223]:

```
(array([0.86306691, 1.01415927, 1.32009698]),
 array([[ 0.44047882,  0.41054478, -0.79839301],
        [ 0.89438   , -0.1235395 ,  0.42990977],
        [ 0.07786414, -0.90343289, -0.42159957]]))
```

Row wise and column wise operation

numpy can perform row wise and column wise operation

In [224]:

```
# Processing array. Generating random numbers
from numpy import random
m = random.rand(3,3)
m
```

Out[224]:

```
array([[0.09963303, 0.83013549, 0.55174146],
       [0.86678417, 0.21088445, 0.01877283],
       [0.22974397, 0.22464909, 0.76521392]])
```

Column wise operation

In [225]:

```
# max in each column
m.max(axis=0)
```

Out[225]:

```
array([0.86678417, 0.83013549, 0.76521392])
```

Row wise operation

In [226]:

```
# max in each row
m.max(axis=1)
```

Out[226]:

```
array([0.83013549, 0.86678417, 0.76521392])
```

In [227]:

```
# Replace the value in the array
a = np.array([1,2,3,4,5])
a[3] = 30
a
```

Out[227]:

```
array([ 1,  2,  3, 30,  5])
```

Random Numbers

There are functions in python that generates random numbers.
`random.rand` creates a random number from 0 to 1 i.e. uniform random nos in the interval [0,1]

In [228]:

```
from numpy import random
random.rand(3,3)
```

Out[228]:

```
array([[0.90239891, 0.65759981, 0.02448207],
       [0.75593925, 0.14397116, 0.47867933],
       [0.68883796, 0.26495613, 0.44628354]])
```

Random Number is also be generated using numpy random randn function

In [229]:

```
# Random numbers can be generated using the randn function.
# These generates the random numbers
import random
#eigen values & eigen vectors
x = np.random.randn(10,3)
x
```

Out[229]:

```
array([[-1.88990486e-01,  1.29066040e+00,  2.99879038e-01],
       [-5.02731804e-01,  7.57457444e-01, -7.20859414e-01],
       [ 2.93787529e-01, -3.14468073e-03, -9.78333955e-01],
       [-1.58497327e-01, -9.80110688e-01,  1.38220066e-03],
       [-1.36772463e+00,  9.18363294e-01,  1.50199306e+00],
       [ 6.20667906e-01, -5.53588933e-02,  8.57904724e-01],
       [ 3.58693866e-01, -6.31519598e-01,  6.57580842e-01],
       [-1.43974830e+00, -6.94972577e-02, -2.75924518e-01],
       [-2.00373312e-02,  5.79524951e-01,  6.13074242e-01],
       [-9.69759284e-01, -6.38124496e-01,  5.90714692e-01]])
```

In [230]:

```
### arange Arranging in acscending order
np.arange(1, 11)
```

Out[230]:

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [231]:

```
### linspace
#It generates equal interval values from 1 to 10
lin = np.linspace(1.0, 10, num=10)
lin
```

Out[231]:

```
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

In [232]:

```
### logspace
#It generates values from from 10^3 to 10^4
log1 = np.logspace(3.0, 4.0, num=4)
log1
```

Out[232]:

```
array([ 1000.        ,  2154.43469003,  4641.58883361, 10000.
])
```

In [233]:

```
### hstack & vstack
f = np.array([1,2,3])
g = np.array([4,5,6])

# Horizontal stack
np.hstack((f, g))
```

Out[233]:

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

In [234]:

```
# Vertical stack  Adding in rows
np.vstack((f, g))
```

Out[234]:

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

String operations

Numpy provides a set of vectorized string operations for arrays of type numpy.string *or numpy.unicode*. Here are some character functions

In [235]:

```
x= "Hello India"
```

In [236]:

```
# To upper
np.char.upper(x)
```

Out[236]:

```
array('HELLO INDIA', dtype='<U11')
```

In [237]:

```
# To lower
np.char.lower(x)
```

Out[237]:

```
array('hello india', dtype='<U11')
```

In [238]:

```
# Splitting a char
np.char.split('Hello, Hi, How', sep = ',')
```

Out[238]:

```
array(list(['Hello', ' Hi', ' How']), dtype=object)
```

In [239]:

```
# Join
np.char.join(['-', ':'], ['Goodday', ' all'])
```

Out[239]:

```
array(['G-o-o-d-d-a-y', ' :a:l:l'], dtype='<U13')
```

In [240]:

```
#counting a substring
a=np.array(['Hi', 'How', 'How'])
np.char.count(a, 'How')
```

Out[240]:

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

Below listed are few functions of numpy string

Function	Description
numpy.capitalise()	capitalises the function
numpy.index()	return index of selected string
numpy.isalpha())	return true if all characters are are alphabets
numpy.isdecimal()	return true if all characters are decimal
numpy.greater_equal()	return true if string1 >= string2 or not
numpy.less_equal()	return true if string1 is <= string2 or not.

In [241]:

```
# Capitalize
import numpy as np
x= "new delhi"

np.char.capitalize(x)
```

Out[241]:

```
array('New delhi', dtype='<U9')
```

In [242]:

```
# Adding two string
import numpy as np
x= "New"
y = "Delhi"

np.char.add(x,y)
```

Out[242]:

```
array('NewDelhi', dtype='<U8')
```

In [243]:

```
# Returns the index
x= "Today is good day"
np.char.index(x,"is")
```

Out[243]:

```
array(6)
```

In [244]:

```
#Returns true if all the character are alphabets
x= "T122343"
np.char.isalpha(x)
```

Out[244]:

```
array(False)
```

In [245]:

```
#Returns true if all the character are decimal
x= "10"
np.char.isdecimal(x)
```

Out[245]:

```
array(True)
```

In [246]:

```
#Returns true if all the character are present but there can be some
x= "madras"
y = "mad"
np.char.greater_equal(x,y)
```

Out[246]:

```
array(True)
```

In [247]:

```
#Returns true if all the character are equal but there can be some
x= "del"
y = "delhi"
np.char.less_equal(x,y)
```

Out[247]:

```
array(True)
```

Saving the Array to files

In [248]:

```
# Saving arrays to files and loading from files

arr = np.arange(10)
np.save('filename',arr)

#loading from file
a = np.load('filename.npy')
```

In [249]:

```
#save as zip file
np.savez('filename.npz', x=arr, y=a)

#Loading the file
newarr = np.load('filename.npz')
print(newarr['x'])
print(newarr['y'])
```

```
[0 1 2 3 4 5 6 7 8 9]
[0 1 2 3 4 5 6 7 8 9]
```

In [250]:

```
#saving as text file
np.savetxt('filename.txt', arr)
np.savetxt('filename.txt', arr, delimiter = ',')
narr = np.loadtxt('filename.txt', delimiter = ',')
```

Pandas

Pandas is an open-source, BSD-licensed Python library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

The Pandas module is a high performance, highly efficient, and high level data analysis library.

Where Pandas are used

- Import data
- Clean the data
- Exploring data, gain insight into data
- Preparing the data for analysis
- Analysing the data

Series
Series is a one-dimensional labeled array which can be of any data type (integer, string, float, python objects, etc.). The axis labels are collectively called index.
General Syntax

```
pandas.Series( data, index, dtype, copy)
```

In [251]:

```
import pandas as pd
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print(s)
```

```
0    a
1    b
2    c
3    d
dtype: object
```

In [252]:

```
import pandas as pd
from pandas import Series, DataFrame

obj = Series([1,2,3,4,5,6])

print(obj)

print(obj.values)

print(obj.index)
```

```
0    1
1    2
2    3
3    4
4    5
5    6
dtype: int64
[1 2 3 4 5 6]
RangeIndex(start=0, stop=6, step=1)
```

In [253]:

```
# Filtering or subsetting can be done in series
mymarks = Series([178,200,199,197], index=['maths', 'chemistry', 'biology', 'physics'])
mymarks
```

Out[253]:

```
maths        178
chemistry    200
biology      199
physics      197
dtype: int64
```

In [254]:

```
# Taking via row index
mymarks['maths']
```

Out[254]:

```
178
```

In [255]:

```
#Filtering
mymarks[mymarks > 180]
```

Out[255]:

```
chemistry    200
biology      199
physics      197
dtype: int64
```

In [256]:

```
case1 = 'maths' in mymarks  # returns true
print(case1)
case2 ='english' in mymarks # Returns false

print(case2)
```

```
True
False
```

In [257]:

```
#convert Series to dictionary
mymarks_dict = mymarks.to_dict()
print(mymarks_dict)
```

```
{'maths': 178, 'chemistry': 200, 'biology': 199, 'physics': 197}
```

In [258]:

```
#convert dict to Series
mymarks_ser = Series(mymarks_dict)
print(mymarks_ser)
```

```
maths        178
chemistry    200
biology      199
physics      197
dtype: int64
```

Checking for null and not null

In [259]:

```
newsubs = ['biology','chemistry','maths', 'physics', 'english']
newvals = Series(mymarks_dict, index=newsubs)
newvals
```

Out[259]:

```
biology      199.0
chemistry    200.0
maths        178.0
physics      197.0
english        NaN
dtype: float64
```

In [260]:

```
# Checking for null
pd.isnull(newvals)
```

Out[260]:

```
biology      False
chemistry    False
maths        False
physics      False
english       True
dtype: bool
```

In [261]:

```
# Checking for not null
pd.notnull(newvals)
```

Out[261]:

```
biology       True
chemistry     True
maths         True
physics       True
english      False
dtype: bool
```

In [262]:

```
# addition happens based on index values
mymarks + newvals
newvals.name = 'My Public results'
newvals.index.name = 'Marks'
newvals
```

Out[262]:

```
Marks
biology      199.0
chemistry    200.0
maths        178.0
physics      197.0
english        NaN
Name: My Public results, dtype: float64
```

In [263]:

```
#More indexing options
newdf = Series([1,2,3,4], index=['A','B','C', 'D'])
newdf
```

Out[263]:

```
A    1
B    2
C    3
D    4
dtype: int64
```

In [264]:

```
# Getting all index values
myindex =newdf.index
myindex
```

Out[264]:

```
Index(['A', 'B', 'C', 'D'], dtype='object')
```

In [265]:

```
myindex[2]
```

Out[265]:

```
'C'
```

In [266]:

```
#Direct modification of a index in not possible, error
myindex[2] = 'C1'
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-266-94d4f6b3af02> in <module>
      1 #Direct modification of a index in not possible, error
----> 2 myindex[2] = 'C1'

/usr/local/lib/python3.6/dist-packages/pandas/core/indexes/base.py i
n __setitem__(self, key, value)
   3936
   3937     def __setitem__(self, key, value):
-> 3938         raise TypeError("Index does not support mutable oper
ations")
   3939
   3940     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

Reindexing

Reindexing changes the row labels and column labels of a DataFrame.

In [267]:

```
newdf = Series([1,2,3,4], index=['A','B','C', 'D'])
newdf1 = newdf.reindex(['A','B','C','D','F','E','z'])
newdf1
```

Out[267]:

```
A    1.0
B    2.0
C    3.0
D    4.0
F    NaN
E    NaN
z    NaN
dtype: float64
```

In [268]:

```
newdf1.reindex(['A','B','C','D','F','E','z','N'], fill_value=0)
newdf1
```

Out[268]:

```
A    1.0
B    2.0
C    3.0
D    4.0
F    NaN
E    NaN
z    NaN
dtype: float64
```

In [269]:

```
# Creating a series
newdf2 = Series(['India', 'China', 'Malaysia'], index=[0,5,10])
newdf2
```

Out[269]:

```
0        India
5        China
10    Malaysia
dtype: object
```

In [270]:

```
ser1 = Series(np.arange(3),index=['A','B','C'])
ser1 = 2*ser1
ser1
```

Out[270]:

```
A    0
B    2
C    4
dtype: int64
```

In [271]:

```
# Selecting by row name
ser1['B']
```

Out[271]:

```
2
```

In [272]:

```
# Selecting by index number
ser1[1]
```

Out[272]:

```
2
```

In [273]:

```
# Selecting multiple rows numbers

ser1[0:3]
```

Out[273]:

```
A    0
B    2
C    4
dtype: int64
```

In [274]:

```
# Selecting multiple rows names
ser1[['A','B','C']]
```

Out[274]:

```
A    0
B    2
C    4
dtype: int64
```

DATAFRAME:

DataFrame is the widely used data structure of pandas. DataFrame can be used with two dimensional arrays. DataFrame has two different index i.e.column-index and row-index.

How to create a dataframe

DataFrame can be created by following

- Creating from a collection or list
- Importing data from file or database

In [275]:

```
# Creating a simple dataframe from a list
import pandas as pd
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print(df)
```

```
   0
0  1
1  2
2  3
3  4
4  5
```

In [276]:

```
# Creating dataframe from List for multiple columns

data = [['Roger',10],['Andy',12],['Rafael',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
print(df)
```

```
     Name  Age
0   Roger   10
1    Andy   12
2  Rafael   13
```

Pandas Operation

One of the essential pieces of NumPy is the ability to perform quick elementwise operations, both with basic arithmetic (addition, subtraction, multiplication, etc.) trigonometric functions, exponential and logarithmic functions, etc. Pandas inherits most of the functionality from NumPy

In [277]:

```
# removing all the warnings
import warnings
warnings.filterwarnings('ignore')
```

In [278]:

```
# Loading the in build dataset
import seaborn.apionly as sns
data = sns.load_dataset('titanic')
data.head()
```

Out[278]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_ma
0	0	3	male	22.0	1	0	7.2500	S	Third	man	Tr
1	1	1	female	38.0	1	0	71.2833	C	First	woman	Fal
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	Fal
3	1	1	female	35.0	1	0	53.1000	S	First	woman	Fal
4	0	3	male	35.0	0	0	8.0500	S	Third	man	Tr

Selecting a column

In [279]:

```
# Indexing a dataframe, selecting a column
data["fare"].head()
```

Out[279]:

```
0     7.2500
1    71.2833
2     7.9250
3    53.1000
4     8.0500
Name: fare, dtype: float64
```

Removing a column

In [280]:

```
# Removing a column
# using del function
print ("Deleting the last column using DEL function:")
del data['alone']
data.head()
```

```
Deleting the last column using DEL function:
```

Out[280]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_ma
0	0	3	male	22.0	1	0	7.2500	S	Third	man	Tr
1	1	1	female	38.0	1	0	71.2833	C	First	woman	Fal
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	Fal
3	1	1	female	35.0	1	0	53.1000	S	First	woman	Fal
4	0	3	male	35.0	0	0	8.0500	S	Third	man	Tr

Slicing

Slicing is a computationally fast way to methodically access parts of your data.

slicing by columns
loc uses string indices; iloc uses integers

In [281]:

```
# Getting only selected columns use command loc
data1= data.loc[:,['sex','age','fare']]
data1.head()
```

Out[281]:

	sex	age	fare
0	male	22.0	7.2500
1	female	38.0	71.2833
2	female	26.0	7.9250
3	female	35.0	53.1000
4	male	35.0	8.0500

In [282]:

```
# Slicing via column index   use command iloc
dat2 = data.iloc[:,[1,2,3]]
dat2.head()
```

Out[282]:

	pclass	sex	age
0	3	male	22.0
1	1	female	38.0
2	3	female	26.0
3	1	female	35.0
4	3	male	35.0

In [283]:

```
# selecting a range of index
data4 = data.iloc[:, 0:3]
data4.head()
```

Out[283]:

	survived	pclass	sex
0	0	3	male
1	1	1	female
2	1	3	female
3	1	1	female
4	0	3	male

Slicing by rows

In [284]:

```
# The below given example takes 3 rows
dat5=data.loc[1:3]
dat5
```

Out[284]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_ma
1	1	1	female	38.0	1	0	71.2833	C	First	woman	Fal
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	Fal
3	1	1	female	35.0	1	0	53.1000	S	First	woman	Fal

In [285]:

```
#select the first 5 rows (rows 0,1,2,3,4)
df=data[:5]
df
```

Out[285]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_ma
0	0	3	male	22.0	1	0	7.2500	S	Third	man	Tr
1	1	1	female	38.0	1	0	71.2833	C	First	woman	Fal
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	Fal
3	1	1	female	35.0	1	0	53.1000	S	First	woman	Fal
4	0	3	male	35.0	0	0	8.0500	S	Third	man	Tr

In [286]:

```
# Selecting 0,2,4 row
data.loc[[0,2,4]]
```

Out[286]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male
0	0	3	male	22.0	1	0	7.250	S	Third	man	True
2	1	3	female	26.0	0	0	7.925	S	Third	woman	False
4	0	3	male	35.0	0	0	8.050	S	Third	man	True

Slicing by Rows and Columns

In [287]:

```
# Example to take rows 0,1,2 and columns 1,2
data.iloc[0:3, 1:3]
```

Out[287]:

	pclass	sex
0	3	male
1	1	female
2	3	female

In [288]:

```
# Example to take rows 0,1,2 and all columns
data.iloc[0:3, ]
```

Out[288]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_ma
0	0	3	male	22.0	1	0	7.2500	S	Third	man	Tr
1	1	1	female	38.0	1	0	71.2833	C	First	woman	Fal
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	Fal

Manipulating the Datasets

Creating frequency distribution for categorical variable

In [289]:

```
data["class"].value_counts(ascending=True)
```

Out[289]:

```
Second    184
First     216
Third     491
Name: class, dtype: int64
```

In [290]:

```
#Crosstab
#A crosstab creates a bivariate frequency distribution.

pd.crosstab(data.sex,data.alive)
```

Out[290]:

alive	no	yes
sex		
female	81	233
male	468	109

Continous variables

In [291]:

```
# Getting mean
data["fare"].mean()
```

Out[291]:

```
32.204207968574636
```

In [292]:

```
# Getting ssum
data["fare"].sum()
```

Out[292]:

```
28693.9493
```

In [293]:

```
# Getting 90th percentile value
data["fare"].quantile(0.9)
```

Out[293]:

```
77.9583
```

In [294]:

```
## summary of a continuous variable
data["fare"].describe()
```

Out[294]:

```
count    891.000000
mean      32.204208
std       49.693429
min        0.000000
25%        7.910400
50%       14.454200
75%       31.000000
max      512.329200
Name: fare, dtype: float64
```

In [295]:

```
# Creating new variables
data['fare1']=data["fare"] + data["pclass"]
data.head()
```

Out[295]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_ma
0	0	3	male	22.0	1	0	7.2500	S	Third	man	Tr
1	1	1	female	38.0	1	0	71.2833	C	First	woman	Fal
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	Fal
3	1	1	female	35.0	1	0	53.1000	S	First	woman	Fal
4	0	3	male	35.0	0	0	8.0500	S	Third	man	Tr

In [296]:

```
# Sorting a dataframe
data.sort_values('fare',ascending=False).head()
```

Out[296]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult
258	1	1	female	35.0	0	0	512.3292	C	First	woman	
737	1	1	male	35.0	0	0	512.3292	C	First	man	
679	1	1	male	36.0	0	1	512.3292	C	First	man	
88	1	1	female	23.0	3	2	263.0000	S	First	woman	
27	0	1	male	19.0	3	2	263.0000	S	First	man	

Aggregation in Dataframe

Aggregation can be done as it is done using sql. here we use group by function

In [297]:

```
## Groupby Function
data.groupby('sex').fare.min()
```

Out[297]:

```
sex
female    6.75
male      0.00
Name: fare, dtype: float64
```

In [298]:

```
## Group for Multiple
data.groupby('sex').fare.agg(['count','min','max','mean'])
```

Out[298]:

	count	min	max	mean
sex				
female	314	6.75	512.3292	44.479818
male	577	0.00	512.3292	25.523893

In [299]:

```
# Aggregation for categorical variables
data["sex"].value_counts(ascending=True)
```

Out[299]:

```
female    314
male      577
Name: sex, dtype: int64
```

Transform

Transform is same as groupby but the result is applied through all the values in dataframe

In [300]:

```
data['new_fare']=data.groupby('sex')['fare'].transform(sum)
data.head()
```

Out[300]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_ma
0	0	3	male	22.0	1	0	7.2500	S	Third	man	Tr
1	1	1	female	38.0	1	0	71.2833	C	First	woman	Fal
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	Fal
3	1	1	female	35.0	1	0	53.1000	S	First	woman	Fal
4	0	3	male	35.0	0	0	8.0500	S	Third	man	Tr

Higher Order Functions

Functions that take a function as an argument or return a function

Map

`map` functions expects a function object and any number of iterables like list, dictionary, etc. It executes the function_object for each element in the sequence and returns a list of the elements modified by the function object.

Basic syntax

`map(function_object, iterable1, iterable2,...)`

In [4]:

```
import warnings
warnings.filterwarnings('ignore')
```

In [5]:

```
# Loading the in build dataset
import seaborn.apionly as sns
df = sns.load_dataset('titanic')
df.head()
```

Out[5]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_ma
0	0	3	male	22.0	1	0	7.2500	S	Third	man	Tr
1	1	1	female	38.0	1	0	71.2833	C	First	woman	Fal
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	Fal
3	1	1	female	35.0	1	0	53.1000	S	First	woman	Fal
4	0	3	male	35.0	0	0	8.0500	S	Third	man	Tr

Converting all the values of malke to 1 and female to 0 using map

In [302]:

```
df['Sex_num']=df.sex.map({'female':0, 'male':1})
df.head()
```

Out[302]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_ma
0	0	3	male	22.0	1	0	7.2500	S	Third	man	Tr
1	1	1	female	38.0	1	0	71.2833	C	First	woman	Fal
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	Fal
3	1	1	female	35.0	1	0	53.1000	S	First	woman	Fal
4	0	3	male	35.0	0	0	8.0500	S	Third	man	Tr

lambda functions in python

- lambda operator can have any number of arguments
- It can have only one expression.
- It cannot contain any statements
- It returns a function object which can be assigned to any variable.

The basic syntax
`lambda arguments : expression`

Consider the following example without lambda and with lambda

In [303]:

```
# Without lamdba function
def add(x, y):
    return x + y

# Call the function
add(2, 3)  # Output: 5
```

Out[303]:

```
5
```

In [304]:

```
#  With lamda function
add = lambda x, y : x + y
add(2, 3) # Output: 5
```

Out[304]:

```
5
```

In [306]:

```
# Multiple iterables to the map function
list_a = [1, 2, 3]
list_b = [10, 20, 30]

x=map(lambda x, y: x + y,list_a,list_b)
list(x)
```

Out[306]:

```
[11, 22, 33]
```

In [305]:

```
# Applying lamda function to data frame
df.fare.map(lambda x : x *2).head()
```

Out[305]:

```
0     14.5000
1    142.5666
2     15.8500
3    106.2000
4     16.1000
Name: fare, dtype: float64
```

In [7]:

```
# Applying lambda function to  class column
df["classNew"] = df["class"].map(lambda x: x.upper()) # converting entire value
 to upper
df.head()
```

Out[7]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_ma
0	0	3	male	22.0	1	0	7.2500	S	Third	man	Tr
1	1	1	female	38.0	1	0	71.2833	C	First	woman	Fal
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	Fal
3	1	1	female	35.0	1	0	53.1000	S	First	woman	Fal
4	0	3	male	35.0	0	0	8.0500	S	Third	man	Tr

In [11]:

```
# applying sqrt to all the values
df.fare.map(lambda x:np.sqrt(x)).head()
```

Out[11]:

```
0    2.692582
1    8.442944
2    2.815138
3    7.286975
4    2.837252
Name: fare, dtype: float64
```

Apply

Apply the function over the column

apply() can apply a function along any axis of the dataframe

In [307]:

```
# Applying sqrt root to all the values in the dataframe
df.fare.apply(np.sqrt).head()
```

Out[307]:

```
0    2.692582
1    8.442944
2    2.815138
3    7.286975
4    2.837252
Name: fare, dtype: float64
```

Applymap

`applymap()` applies a function to every single element in the entire dataframe

In [8]:

```
# Apply a square root function to every single cell in the whole data frame
# applymap() applies a function to every single element in the entire dataframe.

import numpy as np
df[['fare','age']].applymap(np.sqrt).head()
```

Out[8]:

	fare	age
0	2.692582	4.690416
1	8.442944	6.164414
2	2.815138	5.099020
3	7.286975	5.916080
4	2.837252	5.916080

Difference between map, apply and applymap

map

- map iterates over each element
- map is defined to Series data type
- map is element wise operation for a Series data

df['column1'].map(lambda x: 5+x), this will add 5 to each element of column1.

applymap

- applymap iteartes a function to a entire dataframe.
- applymap is defined to entire dataframe
- applymap is element wise operation for a dataframe data type

apply to multiple column as once
df[['column1','col2,'col3]].applymap(np.sqrt)

apply

- apply iterates over each element in Series and dataframe
- apply is defined to entire series and dataframe
- apply also works elementwise but is suited to more complex operations and aggregation df['column1']].apply(np.sqrt), it will returns the value of sqrt of the column.

Filter

Syntax
`filter(function_object, iterable)`

Filter function expects two arguments, function_object and an iterable. function_object returns a boolean value. function_object is called for each element of the iterable and filter returns only those element for which the function_object returns true

In [309]:

```
#  filter only applied to a dataframe
# Taking only male data
df[df['sex'] == 'male'].head()
```

Out[309]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True
5	0	3	male	NaN	0	0	8.4583	Q	Third	man	True
6	0	1	male	54.0	0	0	51.8625	S	First	man	True
7	0	3	male	2.0	3	1	21.0750	S	Third	child	False

In [310]:

```
# filter applies to a list
a = [1, 2, 3, 4, 5, 6]
b= filter(lambda x : x % 2 == 0, a) # Output: [2, 4, 6]
list(b)
```

Out[310]:

```
[2, 4, 6]
```

In [311]:

```
# Filter applied to a dictionary
dict_a = [{'name': 'Murray', 'rank': 4}, {'name': 'Nadal', 'rank': 1}]
fa= filter(lambda x : x['name'] == 'Murray', dict_a) # Output: [{'name': 'Murray', 'rank': 10}]
list(fa)
```

Out[311]:

```
[{'name': 'Murray', 'rank': 4}]
```

Reduce

The function `reduce(func, seq)` applies the function func() to the sequence seq. It returns a single value.

In [312]:

```
import functools
functools.reduce(lambda x,y: x+y, [47,11,42,13])
```

Out[312]:

```
113
```

In [313]:

```
f = lambda a,b: a if (a > b) else b
f
functools.reduce(f, [47,11,42,102,13]) #o/p: 102
```

Out[313]:

```
102
```

In [314]:

```
functools.reduce(lambda x, y: x+y, range(1,101)) #o/p: 5050
```

Out[314]:

```
5050
```

Join And Merge Pandas Dataframe

Merging two dataframe combines two dataframes or data sets together into one aligning based on common columns.
MERGE is in-memory join operations similar to relational database like SQL

Here we create different data frame for merge. In Python merge and join refers to same things

In [315]:

```
import pandas as pd
df1= pd.DataFrame({
        'id':[1,2,3,4,5],
        'Name': ['Rahul', 'Sachin', 'VVS', 'Saurav', 'Anil'],
        'role':['Batsman','All rounder','Batsman','All rounder','Bowler']})
df1
```

Out[315]:

	id	Name	role
0	1	Rahul	Batsman
1	2	Sachin	All rounder
2	3	VVS	Batsman
3	4	Saurav	All rounder
4	5	Anil	Bowler

In [316]:

```
df2= pd.DataFrame({
        'id':[1,2,3,4,5],
        'State': ['Karnataka', 'Maharashtra', 'Andhra Pradesh', 'West Bengal',
'Karnataka']})

df2
```

Out[316]:

	id	State
0	1	Karnataka
1	2	Maharashtra
2	3	Andhra Pradesh
3	4	West Bengal
4	5	Karnataka

In [317]:

```
df3 =pd.merge(df1,df2,on='id')
df3
```

Out[317]:

	id	Name	role	State
0	1	Rahul	Batsman	Karnataka
1	2	Sachin	All rounder	Maharashtra
2	3	VVS	Batsman	Andhra Pradesh
3	4	Saurav	All rounder	West Bengal
4	5	Anil	Bowler	Karnataka

Consider the two dataframes. df_left and df_right for performing join operation

In [318]:

```
import pandas as pd
df_left = pd.DataFrame({
'id':[1,2,3,4,5],
'Name': ['Johnny', 'George', 'Cook', 'Remo', 'Mike'],
'subject_id':['History','Maths','Social','French','English']})
df_left
```

Out[318]:

	id	Name	subject_id
0	1	Johnny	History
1	2	George	Maths
2	3	Cook	Social
3	4	Remo	French
4	5	Mike	English

In [319]:

```
df_right = pd.DataFrame(
    {'id':[1,2,3,4,5],
    'Name': ['gates', 'Brian', 'Bran', 'Bryce', 'Betty'],
    'subject_id':['Maths','Social','Science','French','English']})
df_right
```

Out[319]:

	id	Name	subject_id
0	1	gates	Maths
1	2	Brian	Social
2	3	Bran	Science
3	4	Bryce	French
4	5	Betty	English

In [320]:

```
# left Join
pd.merge(df_left, df_right, on='subject_id', how='left')
```

Out[320]:

	id_x	Name_x	subject_id	id_y	Name_y
0	1	Johnny	History	NaN	NaN
1	2	George	Maths	1.0	gates
2	3	Cook	Social	2.0	Brian
3	4	Remo	French	4.0	Bryce
4	5	Mike	English	5.0	Betty

In [321]:

```
# Right join
pd.merge(df_left, df_right, on='subject_id', how='right')
```

Out[321]:

	id_x	Name_x	subject_id	id_y	Name_y
0	2.0	George	Maths	1	gates
1	3.0	Cook	Social	2	Brian
2	4.0	Remo	French	4	Bryce
3	5.0	Mike	English	5	Betty
4	NaN	NaN	Science	3	Bran

In [322]:

```
# Outer join
pd.merge(df_left, df_right, how='outer', on='subject_id')
```

Out[322]:

	id_x	Name_x	subject_id	id_y	Name_y
0	1.0	Johnny	History	NaN	NaN
1	2.0	George	Maths	1.0	gates
2	3.0	Cook	Social	2.0	Brian
3	4.0	Remo	French	4.0	Bryce
4	5.0	Mike	English	5.0	Betty
5	NaN	NaN	Science	3.0	Bran

In [323]:

```
# Inner Join
pd.merge(df_left, df_right, on='subject_id', how='inner')
```

Out[323]:

	id_x	Name_x	subject_id	id_y	Name_y
0	2	George	Maths	1	gates
1	3	Cook	Social	2	Brian
2	4	Remo	French	4	Bryce
3	5	Mike	English	5	Betty

Data Cleaning

In [324]:

```
# Data cleaning
from pandas import DataFrame
# Creating data frame with NA
import numpy as np
import pandas as pd
dframe = DataFrame([[1,2,3],[np.nan,5,6],[7,np.nan,9],[np.nan,np.nan,np.nan]])
dframe
```

Out[324]:

	0	1	2
0	1.0	2.0	3.0
1	NaN	5.0	6.0
2	7.0	NaN	9.0
3	NaN	NaN	NaN

In [325]:

```
# Dropping NA
clean_dframe = dframe.dropna()
clean_dframe
```

Out[325]:

	0	1	2
0	1.0	2.0	3.0

In [326]:

```
# Dropping the values with all NA
dframe.dropna(how='all')
```

Out[326]:

	0	1	2
0	1.0	2.0	3.0
1	NaN	5.0	6.0
2	7.0	NaN	9.0

In [55]:

```
import pandas as pd
# Creating a dataframe with NA
dframe2 = pd.DataFrame([[1,2,3,np.nan],[2,np.nan,5,6],[np.nan,7,np.nan,9],[1,np.
nan,np.nan,np.nan]])
print("Original dataframe")
print(dframe2)
```

```
Original dataframe
     0    1    2    3
0  1.0  2.0  3.0  NaN
1  2.0  NaN  5.0  6.0
2  NaN  7.0  NaN  9.0
3  1.0  NaN  NaN  NaN
```

In [57]:

```
df2=dframe2.dropna(thresh=2)  # Removing the values which does not have minimum
 of 2 non NA values
print("removing thresold of 2 non NA")
print(df2)
```

```
removing thresold of 2 non NA
     0    1    2    3
0  1.0  2.0  3.0  NaN
1  2.0  NaN  5.0  6.0
2  NaN  7.0  NaN  9.0
```

In [58]:

```
df3=dframe2.dropna(thresh=3)  # Removing values  which does not have minimum of
3 non NA values
print("removing thresold of 3 non NA")
print(df3)
```

```
removing thresold of 3 non NA
     0    1    2    3
0  1.0  2.0  3.0  NaN
1  2.0  NaN  5.0  6.0
```

In [328]:

```
# Creating a dataframe with NA
dframe2 = DataFrame([[1,2,3,np.nan],[2,np.nan,5,6],[np.nan,7,np.nan,9],[1,np.nan
,np.nan,np.nan]])
print("Original dataframe")
print(dframe2)
```

```
Original dataframe
     0    1    2    3
0  1.0  2.0  3.0  NaN
1  2.0  NaN  5.0  6.0
2  NaN  7.0  NaN  9.0
3  1.0  NaN  NaN  NaN
```

In [329]:

```
dframe2.fillna({0:0,1:1,2:2,3:3})

print(dframe2)

dframe2.fillna(0,inplace=True)   # Fill Na with zero

dframe2
```

```
     0    1    2    3
0  1.0  2.0  3.0  NaN
1  2.0  NaN  5.0  6.0
2  NaN  7.0  NaN  9.0
3  1.0  NaN  NaN  NaN
```

Out[329]:

	0	1	2	3
0	1.0	2.0	3.0	0.0
1	2.0	0.0	5.0	6.0
2	0.0	7.0	0.0	9.0
3	1.0	0.0	0.0	0.0

Reshaping Pandas

Reshaping is done using stack and unstack function.

`stack()` function in pandas converts the data into stacked format .i.e. the column is stacked row wise. When more than one column header is present then it stack the specific column header by specified the level.

`unstack()` function in pandas converts the data into unstacked format

In [330]:

```
import pandas as pd
import numpy as np

header = pd.MultiIndex.from_product([['2017','2018'],['IPL','Ranji']])
runs=([[212,145,267,156],[278,189,145,167],[345,267,189,390],[167,144,156,355]])
df = pd.DataFrame(runs,
                  index=['Virat','Rohit','Sachin','Ganguly'],
                  columns=header)
df
```

Out[330]:

	2017		2018	
	IPL	Ranji	IPL	Ranji
Virat	212	145	267	156
Rohit	278	189	145	167
Sachin	345	267	189	390
Ganguly	167	144	156	355

In [331]:

```
# Getting the index
df.index
```

Out[331]:

```
Index(['Virat', 'Rohit', 'Sachin', 'Ganguly'], dtype='object')
```

In [332]:

```
# Getting the columns
df.columns
```

Out[332]:

```
MultiIndex(levels=[['2017', '2018'], ['IPL', 'Ranji']],
           codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

In [333]:

```
# Stack() Function in dataframe stacks the column to rows at level 1

stacked_df=df.stack()
stacked_df
```

Out[333]:

		2017	2018
Virat	**IPL**	212	267
	Ranji	145	156
Rohit	**IPL**	278	145
	Ranji	189	167
Sachin	**IPL**	345	189
	Ranji	267	390
Ganguly	**IPL**	167	156
	Ranji	144	355

In [334]:

```
# unstack the dataframe row to column
unstacked_df = stacked_df.unstack()
unstacked_df
```

Out[334]:

	2017		2018	
	IPL	**Ranji**	**IPL**	**Ranji**
Virat	212	145	267	156
Rohit	278	189	145	167
Sachin	345	267	189	390
Ganguly	167	144	156	355

In [335]:

```
df['2017']
```

Out[335]:

	IPL	Ranji
Virat	212	145
Rohit	278	189
Sachin	345	267
Ganguly	167	144

More operation on DataFrame

In [336]:

```
df = DataFrame(np.arange(16).reshape(4,4),
                        index=[['a','a','b','b'],[1,2,1,2]],
                        columns=[['Delhi','Delhi','Mum','Chn'],['cold','hot','hot',
'cold']])

df
```

Out[336]:

		Delhi		Mum	Chn
		cold	**hot**	**hot**	**cold**
a	**1**	0	1	2	3
	2	4	5	6	7
b	**1**	8	9	10	11
	2	12	13	14	15

In [337]:

```
# Changing the index names
df.index.names = ['INDEX_1','INDEX_2']
df
```

Out[337]:

		Delhi		Mum	Chn
		cold	**hot**	**hot**	**cold**
INDEX_1	**INDEX_2**				
a	**1**	0	1	2	3
	2	4	5	6	7
b	**1**	8	9	10	11
	2	12	13	14	15

In [338]:

```
df.columns.names = ['Cities','Temp']
df
```

Out[338]:

	Cities	Delhi		Mum	Chn
	Temp	cold	hot	hot	cold
INDEX_1	INDEX_2				
a	1	0	1	2	3
	2	4	5	6	7
b	1	8	9	10	11
	2	12	13	14	15

In [339]:

```
df.swaplevel('Cities','Temp',axis=1)
```

Out[339]:

	Temp	cold	hot		cold
	Cities	Delhi	Delhi	Mum	Chn
INDEX_1	INDEX_2				
a	1	0	1	2	3
	2	4	5	6	7
b	1	8	9	10	11
	2	12	13	14	15

In [340]:

```
df.sum(level='Temp',axis=1)
```

Out[340]:

	Temp	cold	hot
INDEX_1	INDEX_2		
a	1	3	3
	2	11	11
b	1	19	19
	2	27	27

In [341]:

```
from pandas import Series
# Creating a series
newdf2 = Series(['India', 'China', 'Malaysia'], index=[0,5,10])
newdf2
```

Out[341]:

```
0        India
5        China
10    Malaysia
dtype: object
```

In [342]:

```
newrange = range(15)
#Forward fill index
newdf2.reindex(newrange, method='ffill')
```

Out[342]:

```
0        India
1        India
2        India
3        India
4        India
5        China
6        China
7        China
8        China
9        China
10    Malaysia
11    Malaysia
12    Malaysia
13    Malaysia
14    Malaysia
dtype: object
```

In [343]:

```
import numpy as np
import pandas as pd
from pandas import Series, DataFrame
from numpy.random import randn

dframe = DataFrame(randn(25).reshape((5,5)),index=['A','B','D','E','F'],columns=
['col1','col2','col3','col4','col5'])
dframe
dframe2 = dframe.reindex(['A','B','C','D','E','F'])
new_columns = ['col1','col2','col3','col4','col5','col6']

dframe2.reindex(columns=new_columns)
```

Out[343]:

	col1	col2	col3	col4	col5	col6
A	-1.365166	-0.825154	1.089048	-1.854003	-0.911492	NaN
B	1.154105	1.214601	1.373438	0.242721	1.623499	NaN
C	NaN	NaN	NaN	NaN	NaN	NaN
D	0.069095	1.636170	0.702303	1.963557	0.725097	NaN
E	-0.063546	-1.975816	-0.402093	-0.020193	-1.050917	NaN
F	0.160303	-1.191684	0.014791	-0.012077	1.242022	NaN

In [344]:

```
dframe.loc[['A','B','C','D','E','F'],new_columns]
```

Out[344]:

	col1	col2	col3	col4	col5	col6
A	-1.365166	-0.825154	1.089048	-1.854003	-0.911492	NaN
B	1.154105	1.214601	1.373438	0.242721	1.623499	NaN
C	NaN	NaN	NaN	NaN	NaN	NaN
D	0.069095	1.636170	0.702303	1.963557	0.725097	NaN
E	-0.063546	-1.975816	-0.402093	-0.020193	-1.050917	NaN
F	0.160303	-1.191684	0.014791	-0.012077	1.242022	NaN

In [345]:

```
dframe2.drop('C')
dframe2=dframe2.drop('C')
dframe2
```

Out[345]:

	col1	col2	col3	col4	col5
A	-1.365166	-0.825154	1.089048	-1.854003	-0.911492
B	1.154105	1.214601	1.373438	0.242721	1.623499
D	0.069095	1.636170	0.702303	1.963557	0.725097
E	-0.063546	-1.975816	-0.402093	-0.020193	-1.050917
F	0.160303	-1.191684	0.014791	-0.012077	1.242022

In [346]:

```
#axis=0 is default
dframe2.drop('col5',axis=1)
```

Out[346]:

	col1	col2	col3	col4
A	-1.365166	-0.825154	1.089048	-1.854003
B	1.154105	1.214601	1.373438	0.242721
D	0.069095	1.636170	0.702303	1.963557
E	-0.063546	-1.975816	-0.402093	-0.020193
F	0.160303	-1.191684	0.014791	-0.012077

Data import and export

The first step to any data science project is to import the data. Pandas provide a wide range of input/output formats. Few of them are given below

- delimited files
- SQL database
- Excel
- HDFS
- json
- html
- pickle
- sas,
- stata

Note

The file path can be absolute file path or if the file is in the working directory just the file name is sufficient

Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file.

For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv

Loading a csv file

In [347]:

```
# Loading  a csv file
import pandas as pd
data = pd.read_csv('bank-data.csv')
data.head()
```

Out[347]:

	id	age	gender	region	income	married	children	car	save_act	current_
0	ID12101	48	FEMALE	INNER_CITY	17546.0	NO	1	NO	NO	
1	ID12102	40	MALE	TOWN	30085.1	YES	3	YES	NO	Y
2	ID12103	51	FEMALE	INNER_CITY	16575.4	YES	0	YES	YES	Y
3	ID12104	23	FEMALE	TOWN	20375.4	YES	3	NO	NO	Y
4	ID12105	57	FEMALE	RURAL	50576.3	YES	0	NO	YES	

Loading a Json file

In [348]:

```
# Create URL to JSON file (alternatively this can be a filepath)
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/da
ta.json'

# Load the first sheet of the JSON file into a data frame
df = pd.read_json(url, orient='columns')
df.head(n=5)
```

Out[348]:

	integer	datetime	category
0	5	2015-01-01 00:00:00	0
1	5	2015-01-01 00:00:01	0
10	5	2015-01-01 00:00:10	0
11	5	2015-01-01 00:00:11	0
12	8	2015-01-01 00:00:12	0

Loading a Excel file

In [349]:

```
# Load Excel File
filePath = "iris.xlsx"

# Load the first sheet of the Excel file into a data frame
df = pd.read_excel(filePath, sheet_name=0, header=0)

# View the first 5 rows
df.head(5)
```

Out[349]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Loading a text file

In [352]:

```
# Read a text file
filePath = "data.txt"

# Load the text into a data frame
df = pd.read_csv(filePath,sep=";")

df
```

Out[352]:

	id	Name	role
0	1	Rahul	Batsman
1	2	Sachin	All rounder
2	3	VVS	Batsman
3	4	Saurav	All rounder
4	5	Anil	Bowler

Loading any delimited file

In [360]:

```
#Reading a delimited file .

df1 = pd.read_csv('data.csv',sep=",")
df1.head()
```

Out[360]:

	id	Name	role
0	1	Rahul	Batsman
1	2	Sachin	All rounder
2	3	VVS	Batsman
3	4	Saurav	All rounder
4	5	Anil	Bowler

Database Connectivity using Python

Python can be used to connect to most of the databases. It can access database and performance all kind of operations permitted for the user.

Below is the example to connect to mysql database

Install mysql connector

```
pip install mysql-connector
```

Creating a connection to the database. Use the username and password from MySQL database

```
# Creating a connection
import mysql.connector
mydb = mysql.connector.connect(
host="127.0.0.1",
port="3306",
user="root",
passwd="root"
)

# Create Statement in Mysql
sql_Query = "select * from bank.customer"

# Fetching data from database
import pandas as pd
df = pd.read_sql(sql_Query, mydb)
print(type(df))
```

Importing Data from a MongoDB

Install mongodb connector

```
pip install pymongo
```

```
# IMporting the library
from pymongo import MongoClient
# Creating connection Mongodb
client = MongoClient('localhost', 27017)

# Fetching data from database
import pandas as pd
df = pd.DataFrame.from_records(client.admin.inventory.find())  # collecti
on Name with the database
df.head()
```

Data Export

Similar to import, Pandas provide a wide range of option to export the data into various output formats. Few of them are given below

- delimited files
- SQL database
- Excel
- HDFS
- json
- html
- pickle
- sas,
- stata

Wrting a dataframe to a csv file

In [362]:

```
df1.to_csv("data.csv",index =False)  # index - Write row names (index).
```

Wrting a dataframe to a excel file

The excel file name can be .xls or .xlsx

In [363]:

```
df1.to_excel("data.xlsx",index =False)  # index - Write row names (index).
```

Wrting a dataframe to a text file

A dataframe can be written into a text file by giving an argument sep.

In [364]:

```
df1.to_csv("data.txt",sep=";",index =False)  # index - Write row names (index).
```

Wrting a dataframe to a json file

A dataframe can be written into a json file

```
orient : string
Indication of expected JSON string format.

The format of the JSON string

‘split’ : dict like {‘index’ -> [index], ‘columns’ -> [columns], ‘data’ -
> [values]}

‘records’ : list like [{column -> value}, … , {column -> value}]

‘index’ : dict like {index -> {column -> value}}

‘columns’ : dict like {column -> {index -> value}}

‘values’ : just the values array

‘table’ : dict like {‘schema’: {schema}, ‘data’: {data}}.
```

In [365]:

```
df1.to_json("data.json",orient="columns")
```

Writing a dataframe to sql

Write records stored in a DataFrame to a SQL database.

```
# Creating a connection
import mysql.connector
mydb = mysql.connector.connect(
host="127.0.0.1",
port="3306",
user="root",
passwd="root"
)



# This will insert df1 dataframe to mysql table tablename,
df1.to_sql('tablename', con=mydb)
```

```
# This will insert df1 dataframe to mysql table tablename, replace the ta
ble if exists
df1.to_sql('tablename', con=mydb,if_exists="replace")
```

Python object Persist and retrieval

Python objects can be persisted and retrieved for the future use . This could be done using joblib from from sklearn package

In [366]:

```
# Creating a python object
dict_a = [{'name': 'Murray', 'rank': 4}, {'name': 'Nadal', 'rank': 1}]
```

In [367]:

```
#Persist a model
from sklearn.externals import joblib
joblib.dump(dict_a, 'sample.joblib')
```

Out[367]:

```
['sample.joblib']
```

In [368]:

```
# retrieving the object
from sklearn.externals import joblib
sample_retrieved = joblib.load('sample.joblib')
sample_retrieved
```

Out[368]:

```
[{'name': 'Murray', 'rank': 4}, {'name': 'Nadal', 'rank': 1}]
```