Web Front End Technology

Python Part-2

Name……

Class……

Group……..

1 Function in Python

In Python, function is a group of related statements that perform a specific task.
Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.
Furthermore, it avoids repetition and makes code reusable.

1.1 Syntax of Function

```
def function_name(parameters):
    """docstring"""
    statement(s)
```

Above shown is a function definition which consists of following components.
Keyword def marks the start of function header.
A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.

Parameters (arguments) through which we pass values to a function. They are optional.

A colon (:) to mark the end of function header.

Optional documentation string (docstring) to describe what the function does.

One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
An optional return statement to return a value from the function.

Example of a function

```
def greet(name):
    """This function greets to
    the person passed in as
    parameter"""
    print("Hello, " + name + ". Good morning!")
```

How to call a function in python?
Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> greet('Paul')
Hello, Paul. Good morning!
```

Note: Try running the above code into the Python shell to see the output.

Docstring
The first string after the function header is called the docstring and is short for documentation string. It is used to explain in brief, what a function does.

Although optional, documentation is a good programming practice. Unless you can remember what you had for dinner last week, always document your code.

In the above example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines. This string is available to us as __doc__ attribute of the function.

For example:
Try running the following into the Python shell to see the output.

```
>>> print(greet.__doc__)
This function greets to
        the person passed into the
        name parameter
```

The return statement
The return statement is used to exit a function and go back to the place from where it was called.

Syntax of return

```
return [expression_list]
```

This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.

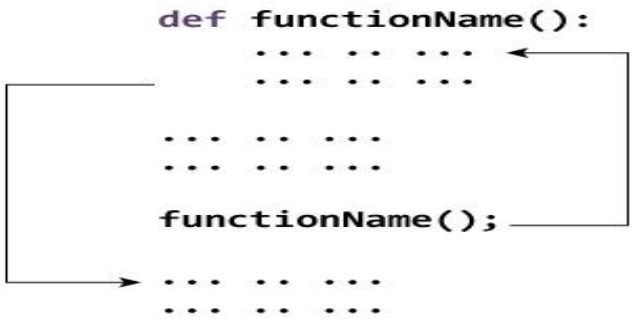
For example:

```
>>> print(greet("May"))
Hello, May. Good morning!
None
```

Here, None is the returned value.
Example of return

```
def absolute_value(num):
        """This function returns the absolute
        value of the entered number"""

        if num >= 0:
                return num
        else:
                return -num

# Output: 2
print(absolute_value(2))

# Output: 4
print(absolute_value(-4))
```

How Function works in Python?



Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.

Lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Here is an example to illustrate the scope of a variable inside a function.

```
def my_func():
    x = 10
    print("Value inside function:",x)

x = 20
my_func()
print("Value outside function:",x)
```

Output

Value inside function: 10
Value outside function: 20

Here, we can see that the value of x is 20 initially. Even though the function my_func()changed the value of x to 10, it did not effect the value outside the function.

This is because the variable x inside the function is different (local to the function) from the one outside. Although they have same names, they are two different variables with different scope.

On the other hand, variables outside of the function are visible from inside. They have a global scope.

We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword global.

1.2 Types of Functions

Basically, we can divide functions into the following two types:

- Built-in functions - Functions that are built into Python.
- User-defined functions - Functions defined by the users themselves.

Python Built-in Function

The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, print() function prints the given object to the standard output device (screen) or to the text stream file.

In Python 3.6 (latest version), there are 68 built-in functions. They are listed below alphabetically along with brief description.

Method	Description
Python abs()	returns absolute value of a number
Python all()	returns true when all elements in iterable is true
Python any()	Checks if any Element of an Iterable is True
Python ascii()	Returns String Containing Printable Representation
Python bin()	converts integer to binary string
Python bool()	Coverts a Value to Boolean
Python bytearray()	returns array of given byte size
Python bytes()	returns immutable bytes object
Python callable()	Checks if the Object is Callable
Python chr()	Returns a Character (a string) from an Integer
Python classmethod()	returns class method for given function
Python compile()	Returns a Python code object
Python complex()	Creates a Complex Number
Python delattr()	Deletes Attribute From the Object
Python dict()	Creates a Dictionary

Method	Description
Python dir()	Tries to Return Attributes of Object
Python divmod()	Returns a Tuple of Quotient and Remainder
Python enumerate()	Returns an Enumerate Object
Python eval()	Runs Python Code Within Program
Python exec()	Executes Dynamically Created Program
Python filter()	constructs iterator from elements which are true
Python float()	returns floating point number from number, string
Python format()	returns formatted representation of a value
Python frozenset()	returns immutable frozenset object
Python getattr()	returns value of named attribute of an object
Python globals()	returns dictionary of current global symbol table
Python hasattr()	returns whether object has named attribute
Python hash()	returns hash value of an object
Python help()	Invokes the built-in Help System
Python hex()	Converts to Integer to Hexadecimal
Python id()	Returns Identify of an Object
Python input()	reads and returns a line of string
Python int()	returns integer from a number or string
Python isinstance()	Checks if a Object is an Instance of Class
Python issubclass()	Checks if a Object is Subclass of a Class
Python iter()	returns iterator for an object
Python len()	Returns Length of an Object

Method	Description
Python list() Function	creates list in Python
Python locals()	returns dictionary of current local symbol table
Python map()	Applies Function and Returns a List
Python max()	returns largest element
Python memoryview()	returns memory view of an argument
Python min()	returns smallest element
Python next()	Retrieves Next Element from Iterator
Python object()	Creates a Featureless Object
Python oct()	converts integer to octal
Python open()	Returns a File object
Python ord()	returns Unicode code point for Unicode character
Python pow()	returns x to the power of y
Python print()	Prints the Given Object
Python property()	returns a property attribute
Python range()	return sequence of integers between start and stop
Python repr()	returns printable representation of an object
Python reversed()	returns reversed iterator of a sequence
Python round()	rounds a floating point number to ndigits places.
Python set()	returns a Python set
Python setattr()	sets value of an attribute of object
Python slice()	creates a slice object specified by range()
Python sorted()	returns sorted list from a given iterable

Method	Description
Python staticmethod()	creates static method from a function
Python str()	returns informal representation of an object
Python sum()	Add items of an Iterable
Python super()	Allow you to Refer Parent Class by super
Python tuple() Function	Creates a Tuple
Python type()	Returns Type of an Object
Python vars()	Returns __dict__ attribute of a class
Python zip()	Returns an Iterator of Tuples
Python __import__()	Advanced Function Called by import

Python User-defined Functions
In this tutorial, you will find the advantages of using user-defined functions and best practices to follow.

What are user-defined functions in Python?
Functions that we define ourselves to do certain specific task are referred as user-defined functions. The way in which we define and call functions in Python are already discussed.

Functions that readily come with Python are called built-in functions. If we use functions written by others in the form of library, it can be termed as library functions.

All the other functions that we write on our own fall under user-defined functions. So, our user-defined function could be a library function to someone else.

Advantages of user-defined functions
User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.

If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.

Programmars working on large project can divide the workload by making different functions.

Example of a user-defined function
Program to illustrate

```
# the use of user-defined functions

def add_numbers(x,y):
  sum = x + y
  return sum

num1 = 5
num2 = 6

print("The sum is", add_numbers(num1, num2))
```

Output

```
Enter a number: 2.4
Enter another number: 6.5
The sum is 8.9
```

Here, we have defined the function `my_addition()` which adds two numbers and returns the result.

This is our user-defined function. We could have multiplied the two numbers inside our function (it's all up to us). But this operation would not be consistent with the name of the function. It would create ambiguity.

It is always a good idea to name functions according to the task they perform.

2 Recursion in Python

Recursion is the process of defining something in terms of itself.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

2.1 Python Recursive Function

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.

Following is an example of recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

Example of recursive function

```
# An example of a recursive function to
# find the factorial of a number

def calc_factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""
```

```
    if x == 1:
        return 1
    else:
        return (x * calc_factorial(x-1))

num = 4
print("The factorial of", num, "is", calc_factorial(num))
```

In the above example, calc_factorial() is a recursive functions as it calls itself.
When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function call multiples the number with the factorial of number 1 until the number is equal to one. This recursive call can be explained in the following steps.

```
calc_factorial(4)            # 1st call with 4
4 * calc_factorial(3)        # 2nd call with 3
4 * 3 * calc_factorial(2)    # 3rd call with 2
4 * 3 * 2 * calc_factorial(1)  # 4th call with 1
4 * 3 * 2 * 1              # return from 4th call as number=1
4 * 3 * 2                  # return from 3rd call
4 * 6                      # return from 2nd call
24                         # return from 1st call
```

Our recursion ends when the number reduces to 1. This is called the base condition.
Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

2.2 Advantages of Recursion

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

2.3 Disadvantages of Recursion

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

Python Program to Display Fibonacci sequence Using Recursion

A Fibonacci sequence is the integer sequence of 0, 1, 1, 2, 3, 5, 8....
The first two terms are 0 and 1. All other terms are obtained by adding the preceding two terms.This means to say the nth term is the sum of $(n-1)^{th}$ and $(n-2)^{th}$ term.

```
def recur_fibo(n):
   """Recursive function to
   print Fibonacci sequence"""
   if n <= 1:
      return n
   else:
      return(recur_fibo(n-1) + recur_fibo(n-2))

# Change this value for a different result
nterms = 10

# uncomment to take input from the user
#nterms = int(input("How many terms? "))

# check if the number of terms is valid
if nterms <= 0:
   print("Plese enter a positive integer")
else:
   print("Fibonacci sequence:")
   for i in range(nterms):
      print(recur_fibo(i))
```

Output

```
Fibonacci sequence:
0 1 1 2 3 5 8 13 21 34
```

Note: To test the program, change the value of nterms.
In this program, we store the number of terms to be displayed in nterms.

A recursive function `recur_fibo()` is used to calculate the nth term of the sequence. We use a `for` loop to iterate and calculate each term recursively.

Python Program to Find Sum of Natural Numbers Using Recursion

```
def recur_sum(n):
   """Function to return the sum
   of natural numbers using recursion"""
   if n <= 1:
      return n
   else:
      return n + recur_sum(n-1)

# change this value for a different result
num = 16

# uncomment to take input from the user
#num = int(input("Enter a number: "))
```

```
if num < 0:
   print("Enter a positive number")
else:
   print("The sum is",recur_sum(num))
```

Output

The sum is 136

3 Exception in python

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error. When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

3.1 Handling an exception

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax

Here is simple syntax of *try....except...else* blocks −

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Here are few important points about the above-mentioned syntax −

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

Example Normal program without Exception handling

a=int(input("Enter any number....."))	Output Enter any number.....y **ValueError**: invalid literal for int() with base 10: 'y'

Example-1 Program with Exception handling

try:
  a=int(input("Enter any number......"))
except:
  print("Exception is generated by python")
	Output
	Enter any number.....y
	Exception is generated by python

Example-2

```
while True:
    try:
       x = int(input("Please enter a number: "))
       break
    except ValueError:
        print("Oops!  That was no valid number.  Try again...")
```

Output

```
Please enter a number: i
Oops!  That was no valid number.  Try again...
Please enter a number:
```

1. First statement between try and except block are executed.
2. If no exception occurs then code under except clause will be skipped.
3. If file don't exists then exception will be raised and the rest of the code in the try block will be skipped
4. When exceptions occurs, if the exception type matches exception name after except keyword, then the code in that except clause is executed.

Example-3

```
import sys

list = ['a', 0, 2, 7]

for entry in list:
    try:
       print("The entry is", entry)
       r = 1/int(entry)
       print("The reciprocal of",entry,"is",r)

    except:
       print("Exception",sys.exc_info()[0],"occured.")
       print("Next entry.")
       print()   #to print new line
```

The *except* Clause with Multiple Exceptions

You can also use the same *except* statement to handle multiple exceptions as follows −

```
try:
   You do your operations here;
   ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
   If there is any exception from the given exception list,
   then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

3.2 The try-finally Clause

You can use a **finally:** block along with a **try:** block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this −

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

You cannot use *else* clause as well along with a finally clause.

Example-4

```
try:
   num1, num2 = eval(input("Enter two numbers, separated by a comma : "))
   result = num1 / num2
   print("Result is", result)

except ZeroDivisionError:
   print("Division by zero is error !!")

except SyntaxError:
   print("Comma is missing. Enter numbers separated by comma like this 1, 2")

except:
   print("Wrong input")

else:
   print("No exceptions")

finally:
   print("This will execute no matter what")
```

Case-1 Output
Enter two numbers, separated by a comma : 7 , y
Wrong input
This will execute no matter what

Case-2 Output
Enter two numbers, separated by a comma : 7 , 0
Division by zero is error !!
This will execute no matter what

Case-3 Output
Enter two numbers, separated by a comma : 7, 2
Result is 3.5
No exceptions
This will execute no matter what

3.3 Raising exceptions

To raise your exceptions from your own methods you need to use raise keyword like this

Syntax
raise ExceptionClass("Your argument")

```
def fun(age):
    if age < 0:
        raise ValueError("Only positive integers are allowed")
     if age % 2 == 0:
        print("age is even")
    else:
        print("age is odd")
 try:
    num = int(input("Enter your age: "))
    fun(num)
 except ValueError:
    print("Only positive integers are allowed")
except:
    print("something is wrong")
```

Output Case-2

Enter your age: -3
Only positive integers are allowed

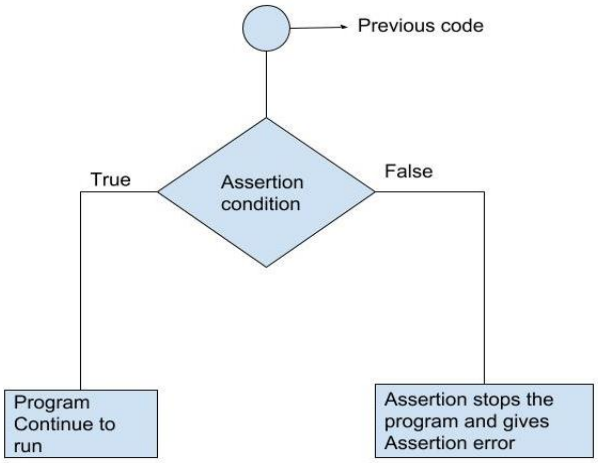
Output Case-3
Enter your age: y
Only positive integers are allowed

4 Assertion in python

Assertions are statements that assert or state a fact confidently in your program. For example, while writing a division function, you're confident the divisor shouldn't be zero, you assert divisor is not equal to zero.

Assertions are simply boolean expressions that checks if the conditions return true or not. If it is true, the program does nothing and move to the next line of code. However, if it's false, the program stops and throws an error.

It is also a debugging tool as it brings the program on halt as soon as any error is occurred and shows on which point of the program error has occurred.



Python assert Statement

Python has built-in assert statement to use assertion condition in the program. assertstatement has a condition or expression which is supposed to be always true. If the condition is false assert halts the program and gives an AssertionError.

Syntax for using Assert in Pyhton:

assert <condition>
assert <condition>,<error message>

In Python we can use assert statement in two ways as mentioned above.
assert statement has a condition and if the condition is not satisfied the program will stop and give AssertionError.
assert statement can also have a condition and a optional error message. If the condition is not satisfied assert stops the program and gives AssertionError along with the error message.
Let's take an example, where we have a function which will calculate the average of the values passed by the user and the value should not be an empty list. We will use assertstatement to check the parameter and if the length is of the passed list is zero, program halts.

Example 1: Using assert without Error Message

def fun(marks):
  assert len(marks) != 0
  return sum(marks)/len(marks)
 	
mark1 = [300,200]
print("Average of mark1:",fun(mark1))
	def fun(marks):
	  assert len(marks) != 0
	  return sum(marks)/len(marks)
	
	mark1 = []
	print("Average of mark1:",fun(mark1))

Output Average of mark1: 250.0	Output AssertionError:

We got an error as we passed an empty list mark1 to assert statement, the condition became false and assert stops the program and give AssertionError.
Now let's pass another list which will satisfy the assert condition and see what will be our output.

Example 2: Using assert with error message

def fun(marks):
  assert len(marks) != 0,"List is empty."
  return sum(marks)/len(marks)
 	
mark2 = [55,88,78,90,79]
print("Average of mark2:",fun(mark2))
	def fun(marks):
	  assert len(marks) != 0,"List is empty."
	  return sum(marks)/len(marks)
	
	mark1 = []
	print("Average of mark1:",fun(mark1))

Average of mark2: 78.0	AssertionError: List is empty.

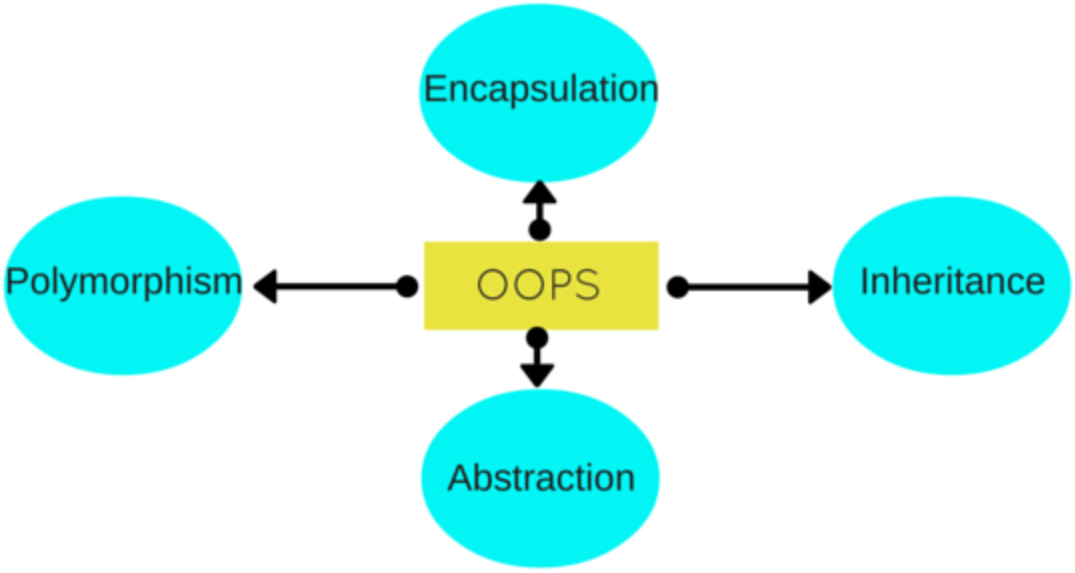
We passed a non-empty list mark2 and also an empty list mark1 to the avg() function and we got output for mark2 list but after that we got an error AssertionError: List is empty. The assert condition was satisfied by the mark2 list and program to continue to run. However, mark1 doesn't satisfy the condition and gives an AssertionError.

Key Points to Remember

- Assertions are the condition or boolean expression which are always supposed to be true in the code.
- assert statement takes an expression and optional message.
- assert statement is used to check types, values of argument and the output of the function.
- assert statement is used as debugging tool as it halts the program at the point where an error occurs.

5 Python - Object Oriented

Object oriented programming – As the name suggests uses objects in programming. Object oriented programming aims to implement real world entities like inheritance, hiding, polymorphism etc in programming. The main aim of OOP is to bind together the data and the functions that operates on them so that no other part of code can access this data except that function.



Class
Here we can take Human Being as a class. A class is a blueprint for any functional entity which defines its properties and its functions. Like Human Being, having body parts, and performing various actions.

Inheritance
Considering HumanBeing a class, which has properties like hands, legs, eyes etc, and functions like walk, talk, eat, see etc. Male and Female are also classes, but most of the properties and functions are included in HumanBeing, hence they can inherit everything from class HumanBeing using the concept of Inheritance.

Objects
My name is Abhishek, and I am an instance/object of class Male. When we say, Human Being, Male or Female, we just mean a kind, you, your friend, me we are the forms of these classes. We have a physical existence while a class is just a logical definition. We are the objects.

Abstraction
Abstraction means, showcasing only the required things to the outside world while hiding the details. Continuing our example, Human Being's can talk, walk, hear, eat, but the details are hidden from the outside world. We can take our skin as the Abstraction factor in our case, hiding the inside mechanism.

Encapsulation
This concept is a little tricky to explain with our example. Our Legs are binded to help us walk. Our hands, help us hold things. This binding of the properties to functions is called Encapsulation.

Polymorphism
Polymorphism is a concept, which allows us to redefine the way something works, by either changing how it is done or by changing the parts using which it is done. Both the ways have different terms for them.

Object-oriented Programming	Procedural Programming
Object-oriented programming is an problem solving approach and used where computation is done by using objects.	Procedural programming uses a list of instructions to do computation step by step.
It makes development and maintenance easier.	In procedural programming, It is not easy to maintain the codes when project becomes lengthy.
It simulates the real world entity. So real world problems can be easily solved through oops.	It doesn't simulate the real world. It works on step by step instructions divided in small parts called functions.
It provides data hiding. so it is more secure than procedural languages. You cannot access private data from anywhere.	Procedural language doesn't provide any proper way for data binding so it is less secure.
Example of object-oriented programming languages are: C++, Java, .Net, Python, C# etc.	Example of procedural languages are: C, Fortran, Pascal, VB etc.

Class − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

Class variable − A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.

Object − A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Data member − A class variable or instance variable that holds data associated with a class and its objects.

Instance variable − A variable that is defined inside a method and belongs only to the current instance of a class.

Inheritance − The transfer of the characteristics of a class to other classes that are derived from it.

Instance − An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.

Instantiation − The creation of an instance of a class.

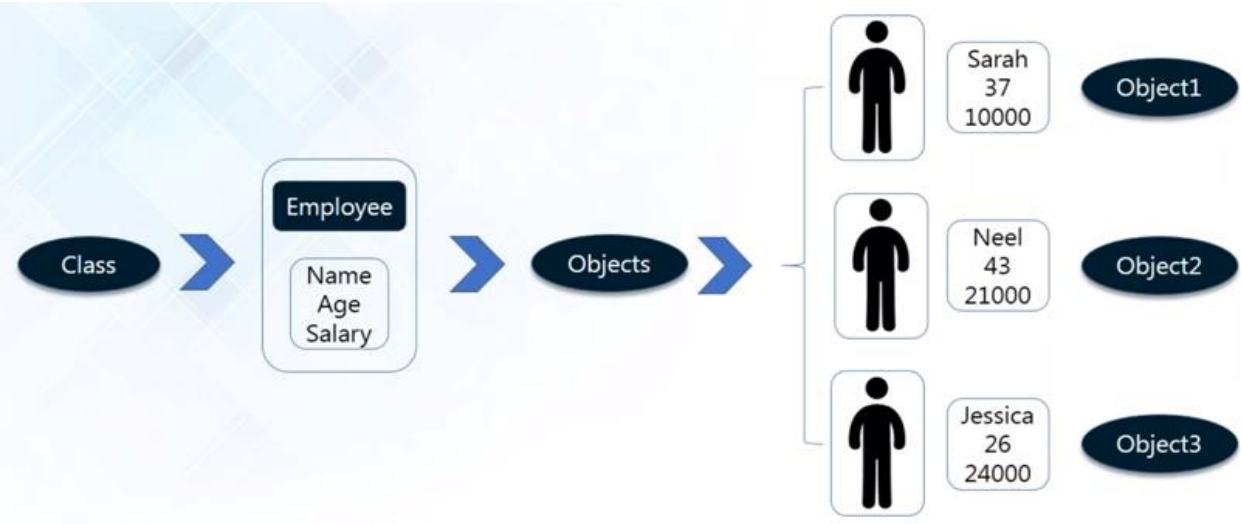
Method − A special kind of function that is defined in a class definition.

Operator overloading − The assignment of more than one function to a particular operator.

5.1 Classes and objects in Python

Python is an object oriented programming language. Unlike procedure oriented programming, where the main emphasis is on functions, object oriented programming stress on objects.
Object is simply a collection of data (variables) and methods (functions) that act on those data. And, class is a blueprint for the object.

We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.
As, many houses can be made from a description, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called **instantiation**.



Defining a Class in Python
Like function definitions begin with the keyword def, in Python, we define a class using the keyword class.
The first string is called docstring and has a brief description about the class. Although not mandatory, this is recommended.

Here is a simple class definition.

```
class MyNewClass:
   '''This is a docstring. I have created a new class'''
   pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.
There are also special attributes in it that begins with double underscores (__). For example, __doc__ gives us the docstring of that class.

As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
class MyClass:
        "This is my second class"
        a = 10
        def func(self):
                print('Hello')
# Output: 10
print(MyClass.a)
```

Output

```
10
<function 0x7feaa932eae8="" at="" myclass.func="">
This is my second class
```

Creating an Object in Python

We saw that the class object could be used to access different attributes.
It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a function call.

>>> ob = MyClass()

This will create a new instance object named *ob*. We can access attributes of objects using the object name prefix.

Attributes may be data or method. Method of an object are corresponding functions of that class. Any function object that is a class attribute defines a method for objects of that class.

This means to say, since MyClass.func is a function object (attribute of class), ob.func will be a method object.

Example-1

```
#how to define class
#class varibales and methods
class man:                      #man is the name of the class
   eyes="black"                 #eyes is variable of the class
   age=25                       #age is variable of the class
   height=157                   #height is variable of the class
```

```
    def prntname(self):                 #prntname is method of the class
        return "kuldeep singh"



#how to create object
obj=man()                                   # creation of object of class man

#how to access the variable and method of class using object
print(obj.eyes)
print(obj.age)
print(obj.prntname())
```

Output

```
black
25
kuldeep singh
```

You may have noticed the *self* parameter in function definition inside the class but, we called the method simply as ***obj.prntname*** () without any arguments. It still worked.

This is because, whenever an object calls its method, the object itself is passed as the first argument. So, ***obj.prntname()*** translates into *man*. ***prntname*** (***obj***).

In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

For these reasons, the first argument of the function in class must be the object itself. This is conventionally called *self*. It can be named otherwise but we highly recommend to follow the convention.

Example-2

```
class student:
    pass
std1=student()
std2=student()

std1.firstname="Amanpreet"
std1.lastname="kaur"
std1.email="kauramanpreet@gmail.com"
std1.marks=75

std2.firstname="Kuldeep Singh"
std2.lastname="sidhu"
std2.email="sidhu.kuldeep89@gmail.com"
std2.marks=55
```

Output

```
Amanpreet
Kuldeep Singh
```

```
print(std1.firstname)
print(std2.firstname)
```

Example-3 Using init method

```
class student:
    def __init__(self,firstname, lastname, email,marks):
        self.f=firstname
        self.l=lastname
        self.e=email
        self.m=marks
std1=student("Kuldeep Singh", "Sidhu", "sidhu.kuldeep89@gmail.com",55)
std2=student("Amanpreet", "Brar", "kauramanpreet@gmail.com",75)
print(std1.f)
print(std2.f)
```

Output

```
Kuldeep Singh
Amanpreet
```

Example-4 Using method

```
class student:
    def __init__(self,firstname, lastname, email,marks):
        self.f=firstname
        self.l=lastname
        self.e=email
        self.m=marks
    def fullname(self):
        return "Ful Name>>>> {} {}" .format(self.f,self.l)

std1=student("Kuldeep Singh", "Sidhu", "sidhu.kuldeep89@gmail.com",55)
std2=student("Amanpreet", "Brar", "kauramanpreet@gmail.com",75)

print(std1.fullname())
print(std2.fullname())
```

Output

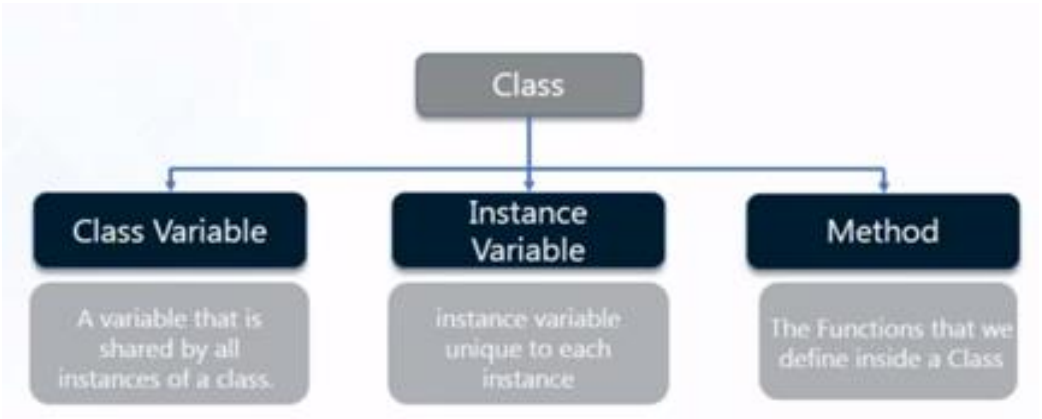
```
Ful Name>>>> Kuldeep Singh Sidhu
Ful Name>>>> Amanpreet Brar
```

Example-5

```
#printing name
class name:
    def makeName(self,nm):
        self.nm=nm
    def showName(self):
        return self.nm
    def prntname(self):
        print("Hello.....", self.nm)
```

```
obj=name()

obj.makeName("Kuldeep Singh")
obj.showName()
obj.prntname()
```

Output
Hello..... Kuldeep Singh



5.2 Python Docstrings

Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods.

It's specified in source code that is used, like a comment, to document a specific segment of code. Unlike conventional source code comments, the docstring should describe what the function does, not how.

Declaring Docstrings: The docstrings are declared using single quotes , "double quotes" or """triple double quotes""" just below the class, method or function declaration. All functions should have a docstring.

Accessing Docstrings: The docstrings can be accessed using the **__doc__** method of the object or using the **help** function.

Example-1

```
def my_function():
    """Demonstrate docstrings and does nothing really."""
    return None
print("Using __doc__:")
```

Output:
Using __doc__:
Demonstrate docstrings and does nothing really.

print(my_function.__doc__) print("Using help:") help(my_function)	**Using help:** Help on function my_function in module __main__: my_function() Demonstrate docstrings and does nothing really.

Example-2

```
class sum:
   "sum of two numbers"
   def __init__(self, a,b):
      "Constructor of class man having two parameter"
      self.num1=a
      self.num2=b
   def sum_two_num(self):
         c=self.num1+self.num2
         print("Sum of two numbers is  ",c)
obj1=sum(5,6)
obj1.sum_two_num()

print("Document string of class sum >>",sum.__doc__)
print("Document string of constructor >>",sum.__init__.__doc__)
```

Output

```
Sum of two numbers is  11
Document string of class sum >> sum of two numbers
Document string of constructor >> Constructor of class man having two parameter
```

5.3 Python Constructors

A constructor is a special type of method (function) which is used to initialize the instance members of the class. Constructor can be parameterized and non-parameterized as well. Constructor definition executes when we create object of the class. Constructors also verify that there are enough resources for the object to perform any start-up task.

Creating a Constructor

A constructor is a class function that begins with double underscore (_). The name of the constructor is always the same __init__().

While creating an object, a constructor can accept arguments if necessary. When we create a class without a constructor, Python automatically creates a default constructor that doesn't do anything.

Every class must have a constructor, even if it simply relies on the default constructor.

Python Constructor Example

Let's create a class named ***sum***, having two functions __init__() function to initialize the variable and ***sum_two_num()*** to display the sum of two numbers.

```
class sum:
    def __init__(self, a,b):
        self.num1=a
        self.num2=b
    def sum_two_num(self):
            c=self.num1+self.num2
            print("Sum of two numbers is  ",c)
obj1=sum(5,6)
obj1.sum_two_num()
```

Output

Sum of two numbers is 11

In Python, Constructors can be parameterized and non-parameterized as well. The parameterized constructors are used to set custom value for instance variables that can be used further in the application.

Python Non Parameterized Constructor Example

```
class Student:
    def __init__(self):
        print("This is non parameterized constructor")
    def show(self,name):
        print("Hello",name)
student = Student()
student.show("kuldeep")
```

Output:

This is non parameterized constructor
Hello kuldeep

Python Parameterized Constructor Example

```
class Student:
    def __init__(self, name):
        print("This is parameterized constructor")
        self.name = name
    def show(self):
        print("Hello",self.name)
student = Student("kuldeep")
student.show()
```

Output:

This is parameterized constructor
Hello kuldeep

5.4 Inheritance in Python

Inheritance is one of the most important aspects of Object Oriented Programming. While programming, many a times, situations arise where we have to write a few classes with some common features and some unique, class-specific features, which include both variables and methods.

In such situations, as per object oriented programming, we can take out the common part and put it in a separate class, and make all the other classes inherit this class, to use its methods and variables, hence reducing re-writing the common features in every class, again and again.

The class which inherits another class is generally known as the **Child class**, while the class which is inherited by other classes is called as the **Parent class**.

Of course, you must only use this for the related classes only, for example, you can define a class **LivingOrganism** with all the basic features of a living organism defined in it, like breathe, eat etc. Now this class can easily be re-used by another class **Animals** and **HumanBeings**, as both of these shares the features.

Also, at times, Inheritance is used to simplify big classes with a lot of variables and methods, into smaller classes by breaking down the functionality into core features and secondary features. The core features are generally kept in the parent class.

Or
Inheritance enable us to define a class that takes all the functionality from parent class and allows us to add more.

Inheritance is a powerful feature in object oriented programming.
It refers to defining a new class with little or no modification to an existing class. The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

Syntax for Inheritance
If we have a class Parent and another class Child and we want the class Child to inherit the class Parent, then

Example-1

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Example-2

```
# Parent class
class Parent:
   # class variable
   a = 10;
   b = 100;
   # some class methods
   def doThis()
   def doThat()

# Child class inheriting Parent class
class Child(Parent):
   # child class variable
   x = 1000;
   y = -1;
   # some child class method
   def doWhat()
   def doNotDoThat()
```

By specifying another class's name in parentheses, while declaring a class, we can specify inheritance. In the example above, all the properties of Parent will be inherited to the Child class. With this, all the methods and variables defined in the class Parent becomes part of Child class too.

Example-3

Let's see a simple python inheritance example where we are using two classes: Animal and Dog. Animal is the parent or base class and Dog is the child class.

Here, we are defining eat() method in Animal class and bark() method in Dog class. In this example, we are creating instance of Dog class and calling eat() and bark() methods by the instance of child class only. Since, parent properties and behaviors are inherited to child object automatically, we can call parent and child class methods by the child instance only.

```
class Animal:                    #parent class
   def eat(self):
     print('Eating...')

class Dog(Animal):               #child class
  def bark(self):                #methods of child class
     print('Barking...')

d=Dog()                    #object creation of child class
d.eat()                    #calling object of parent class from child class
d.bark()                   #calling object from child class
Output:
Eating...
Barking...
```

Benefits of using Inheritance

Here are a few main advantages of using Inheritance in your programs.

- Less code repetition, as the code which is common can be placed in the parent class, hence making it available to all the child classes.
- **Structured Code**: By dividing the code into classes, we can structure our software better by dividing functionality into classes.
- Make the code more scalable.

Accessing Parent Class Element in Child Class

While working in a child class, at some point you may have to use parent class's properties or functions. In order to access parent class's elements you can use the dot . operator.
Parent.variableName

Mentioned above is how you can access the variable, or in case you need to call parent class's function then,

Parent.functionName()

Where Parent is the name of our parent class, and variableName and functionName() are its variable and function respectively.

Below is an example, we have a simple example to demonstrate this:

```
class Parent:
        var1 = 1
        def func1(self):
           # do something here

class Child(Parent):
        var2 = 2
        def func2(self):
     # do something here too
                # time to use var1 from 'Parent'
           myVar = Parent.var1 + 10
           return myVar
```

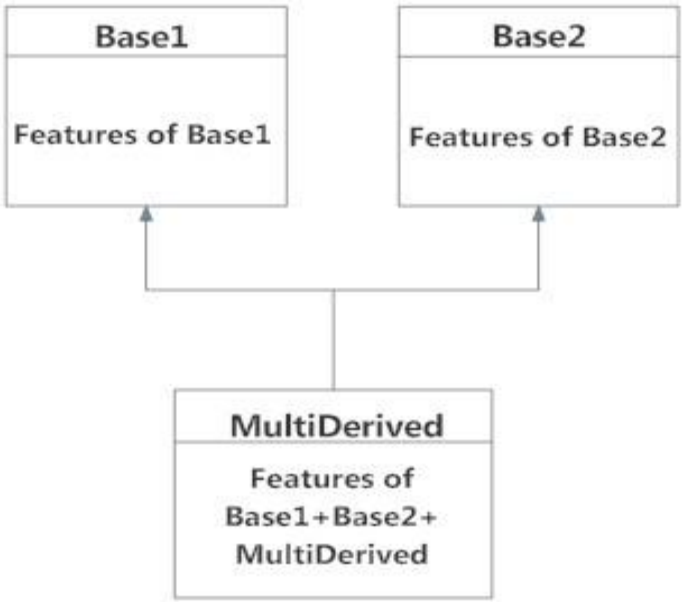
Types of Inheritance in Python

In Python, there are two types of Inheritance:

- Multiple Inheritance
- Multilevel Inheritance

Python - Multiple Inheritance

Multiple Inheritance means that you're inheriting the property of multiple classes into one. In case you have two classes, say A and B, and you want to create a new class which inherits the properties of both A and B, then:



class A:
  # variable of class A
  # functions of class A
 	
class B:
  # variable of class A
  # functions of class A
 	
class C(A, B):
  # class C inheriting property of both class A and B
  # add more properties to class C
	class Base:
	  pass
	
	class Derived1(Base):
	  pass
	
	class Derived2(Derived1):
	  pass
	
	Here, Derived1 is derived from Base, and Derived2 is derived from Derived1.

So just like a child inherits characteristics from both mother and father, in python, we can inherit multiple classes in a single child class.

As you can see, instead of mentioning one class name in parentheses along with the child class, we have mentioned two class names, separated by comma ,. And just to clear your doubts, yes, you can inherit as many classes you want. Therefore, the syntax should actually be:

class A(A1**,** A2**,** A3**, …):**

  # class A inheriting the properties of A1, A2, A3, etc.

    # You can add properties to A class too

```
class Person:                                   #definition of the class starts here
   def __init__(self, personName, personAge):   #defining constructor
      self.name = personName
      self.age = personAge

   def showName(self):                          #defining class methods
      print(self.name)

   def showAge(self):
      print(self.age)                                   #end of class definition

class Student:                              # defining another class
   def __init__(self, studentId):
      self.studentId = studentId

   def getId(self):
      return self.studentId

class Resident(Person, Student):                # extends both Person and Student class
   def __init__(self, name, age, id):
      Person.__init__(self, name, age)
      Student.__init__(self, id)

resident1 = Resident('John', 30, '102')                 # Create an object of the subclass
resident1.showName()
print(resident1.getId())
```

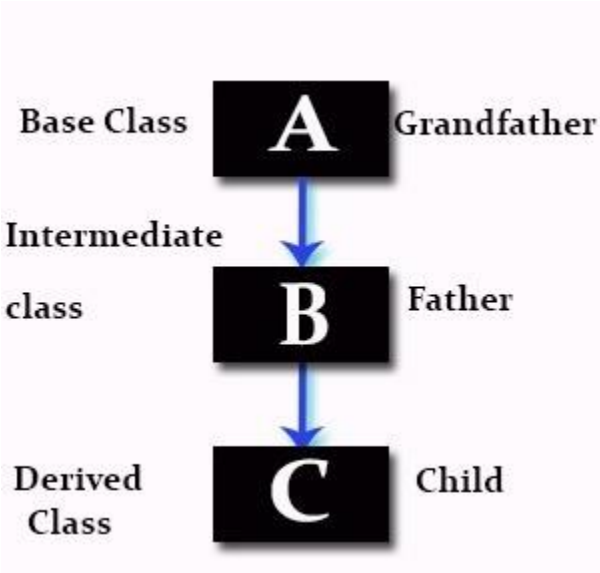
Output

John
102

class father:
  def gardening(self):
  print("I love gardening")
 	
class mother:
  def cooking(self):
  print("I love cooking")
 	
class child(father,mother):
  def sports(self):
  print("I love sports")
 	
obj=child()
obj.gardening()
obj.cooking()
obj.sports()
	Output
	
	I love gardening
	I love cooking
	I love sports

Python - Multilevel Inheritance

In multilevel inheritance, we inherit the classes at multiple separate levels. We have three classes A, B and C, where A is the super class, B is its sub(child) class and C is the sub class of B.



Example-1

```
class A:
    # properties of class A

class B(A):
    # class B inheriting property of class A
    # more properties of class B

class C(B):
    # class C inheriting property of class B
    # thus, class C also inherits properties of class A
    # more properties of class C
```

```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass
```

Here, MultiDerived is derived from classes Base1 and Base2.

Example-2

```
class Animal:
    def eat(self):
        print('Eating...')

class Dog(Animal):
    def bark(self):
        print('Barking...')

class BabyDog(Dog):
    def weep(self):
```

Output:

```
Eating...
Barking...
Weeping
```

```
    print('Weeping...')

d=BabyDog()
d.eat()
d.bark()
d.weep()
```

Example-3

```
class Human:
   being="Human"
   def __init__(self):
      print("Human Class")

class Student:
   status="Student"
   def __init__(self):
      print("Student Class")

class Learner(Human, Student):
   def __init__(self):
      print("Learner Class")

Obj=Learner()
print(Obj.being + " " + " " + Obj.status)
```

Output

```
Learner Class
Human  Student
```

Example-4

```
class father:
   def gardening(self):
      print("I love gardening")

class mother(father):
   def cooking(self):
      print("I love cooking")

class child(mother):
   def sports(self):
      print("I love sports")

obj=child()
obj.gardening()
obj.cooking()
obj.sports()
```

Output

```
I love gardening
I love cooking
I love sports
```

Example-5 All classes have same method name

class father:
  def skills(self):
  print("Gardening, Driving")
 	
class mother:
  def skills(self):
  print("Cooking , Singing")
 	
class child(father,mother):
  def skills(self):
  print("Sports, Swimming")
 	
obj=child()
obj.skills()
	Output
	
	Sports, Swimming

Example-6

Having same name of methods of all base classes if we want to inherit the all methods from all base classes then we need to call parent class from bases class

class father:
  def skills(self):
  print("Gardening, Driving")
 	
class mother:
  def skills(self):
  print("Cooking , Singing")
 	
class child(father,mother):
  def skills(self):
  father.skills(self)
  mother.skills(self)
  print("Sports, Swimming")
 	
obj=child()
obj.skills()
	Output
	
	Cooking , Singing
	Gardening, Driving
	Sports, Swimming

Using issubclass() method

In python, there is a function which helps us to verify whether a particular class is a sub class of another class, that built-in function is issubclass(paramOne, paramTwo), where paramOne and paramTwo can be either class names or class's object name.

```
class Parent:
        var1 = 1
        def func1(self):
```

```
        # do something

class Child(Parent):
        var2 = 2
        def func2(self):
            # do something else
```

In order to check if Child class is a child class of Parent class.
>>> issubclass(Child, Parent)

True

Or using the object of the classes,
Parent p = Parent() #object creation
Child c = Child() #object creation
It's pretty much the same,
>>> issubclass(c, p)

True

6 Information Hiding and Encapsulation in OOP

Information hiding is one of the most important principles of OOP inspired from real life which says that all information should not be accessible to all persons. Private information should only be accessible to its owner.

By Information Hiding we mean "*Showing only those details to the outside world which are necessary for the outside world and hiding all other details from the outside world.*"

Real Life Examples of Information Hiding

- Your name and other personal information is stored in your brain we can't access this information directly. For getting this information we need to ask you about it and it will be up to you how much details you would like to share with us.
- An email server may have account information of millions of people but it will share only our account information with us if we request it to send anyone else accounts information our request will be refused.
- A phone SIM card may store several phone numbers but we can't read the numbers directly from the SIM card rather phone-set reads this information for us and if the owner of this phone has not allowed others to see the numbers saved in this phone we will not be able to see those phone numbers using phone.

In object oriented programming approach we have objects with their attributes and behaviors that are hidden from other classes, so we can say that object oriented programming follows the principle of information hiding.

In the perspective of Object Oriented Programming Information Hiding is,
"Hiding the object details (state and behavior) from the users"

Here by users we mean **"an object"** of another class that is calling functions of this class using the reference of this class object or it may be some other program in which we are using this class.
Information Hiding is achieved in Object Oriented Programming using the following principles,

- All information related to an object is stored within the object
- It is hidden from the outside world
- It can only be manipulated by the object itself

We can achieve information hiding using Encapsulation and Abstraction, so we see these two concepts in detail now.

6.1 Encapsulation

Encapsulation means *"we have enclosed all the characteristics of an object in the object itself".*

In an object oriented python program, you can *restrict access* to methods and variables. This can prevent the data from being modified by accident and is known as *encapsulation*.

Encapsulation is the mechanism for restricting the access to some of an object's components, this means that the internal representation of an object can't be seen from outside of the objects definition.

Encapsulation prevents from accessing accidentally, but not intentionally.
Python does not have the private keyword, unlike some other object oriented languages, but encapsulation can be done.

Instead, it relies on the convention: a class variable that should not directly be accessed should be prefixed with an underscore.

Access Modifiers
Before you can take advantage of encapsulation, you have to understand how Python restricts access to the data stored in variables and methods.

Python has different levels of restriction that control how data can be accessed and from where. Variables and methods can be public, private, or protected. Those designations are made by the number of underscores before the variable or method.

Public
Every variable and method that you've seen so far with the exception of the constructors has been public. Public variables and methods can be freely modified and run from anywhere, either inside or outside of the class. To create a public variable or method, don't use any underscores.

Private
The private designation only allows a variable or method to be accessed from within its own class or object. You cannot modify the value of a private variable from outside of a class. Private variables and methods are preceded by two underscores. Take a look at the example below.
__a=10

Protected

Protected variables and methods are very similar to private ones. You probably won't use protected variables or methods very often, but it's still worth knowing what they are. A variable that is protected can only be accessed by its own class and any classes derived from it. That is more a topic for later, but just be aware that if you are using a class as the basis of another one, protected variables may be a good option. Protected variables begin with a single underscore.

_a=10

To summarize, in Python there are:

Type	**Description**
public methods	Accessible from anywhere
private methods	Accessible only in their own class. starts with two underscores
public variables	Accessible from anywhere
private variables	Accessible only in their own class or by a method if defined. starts with two underscores

Example Using Public Variable

```
class A:
    marks=0     #by default public variable
    name=" "

    def __init__(self):
        self.marks=90
        self.name="Raman"

    def detail(self):
        print(self.name)
        print(self.marks)

obj=A()
print("Printing detail before changing
variable")
obj.detail()
obj.marks=400
obj.name="Suman"
print("Printing detail after changing variable")
obj.detail()
```

```
Printing detail before changing variable
Raman
90

Printing detail after changing variable
Suman
400
```

Example Using Private Variable

```
class A:
   __marks=0
     #defining private variable by _ _ (double
underscore)
   __name=" "

   def __init__(self):
      self.__marks=90
      self.__name="raman"

   def detail(self):
      print(self.__name)
      print(self.__marks)

obj=A()
print("Printing detail before changing variable")
obj.detail()
obj.marks=400
obj.name="Suman"
print("Printing detail after changing variable")
obj.detail()
```

```
Printing detail before changing variable
raman
90

Printing detail after changing variable
raman
90
```

Example Changing private variable

```
class A:
   __marks=0     #defining private variable by _ _
                              (double underscore)
   __name=" "

   def __init__(self):
      self.__marks=90
      self.__name="raman"

   def detail(self):
      print(self.__name)
      print(self.__marks)

   def change(self, m ,n):
      self.__marks=m
      self.__name=n
obj=A()
print("Printing detail before changing variable")
obj.detail()

obj.change(400,"kuldeep")
print("Printing detail after changing variable")
obj.detail()
```

```
Output
Printing detail before changing variable
raman
90

Printing detail after changing variable
kuldeep
400
```

Trying to call private method from outside the class

```
class car:
    def __init__(self):
        pass

    def drive(self):
        print("Driving")
    def __updatesoftware(self):   #private method
        print("Updating Software")

redcar=car()
redcar.drive()
redcar.__updatesoftware()
      #cannot call private method directly or outside
the class
```

Output

```
Driving
AttributeError: 'car' object has no attribut
e '__updatesoftware'
```

Calling private method from inside the class

```
class car:
    def __init__(self):
        self.__updatesoftware()
            #calling private method from inside the class

    def drive(self):
        print("Driving")
    def __updatesoftware(self):   #private method
        print("Updating Software")

redcar=car()
redcar.drive()
```

Output

```
Updating Software
Driving
```

7 Numpy

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

Comparison between Core Python and Numpy

When we say "Core Python", we mean Python without any special modules, i.e. especially without NumPy.

The advantages of Core Python:

- High-level number objects: integers, floating point
- Containers: lists with cheap insertion and append methods, dictionaries with fast lookup

Advantages of using Numpy with Python:

- Array oriented computing
- Efficiently implemented multi-dimensional arrays
- Designed for scientific computation

A Simple Numpy Example

Before we can use NumPy we will have to import it. It has to be imported like any other module:

import numpy

But you will hardly ever see this. Numpy is usually renamed to np:

import numpy as np

We have a list with values, e.g. temperatures in Celsius:

```
cvalues = [20.1, 20.8, 21.9, 22.5, 22.7, 22.3, 21.8, 21.2, 20.9, 20.1]
```

We will turn our list "cvalues" into a one-dimensional numpy array:

```
C = np.array(cvalues)
print(C)
[ 20.1  20.8  21.9  22.5  22.7  22.3  21.8  21.2  20.9  20.1]
```

Let's assume, we want to turn the values into degrees Fahrenheit. This is very easy to accomplish with a numpy array. The solution to our problem can be achieved by simple scalar multiplication:

```
print(C * 9 / 5 + 32)
[ 68.18  69.44  71.42  72.5   72.86  72.14  71.24  70.16  69.62  68.18]
```

The array C has not been changed by this expression:

```
print(C)
[ 20.1  20.8  21.9  22.5  22.7  22.3  21.8  21.2  20.9  20.1]
```

7.1 Arrays

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the *rank* of the array; the *shape* of an array is a tuple of integers giving the size of the array along each dimension.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
import numpy as np
a = np.array([1, 2, 3])                  # Create a rank 1 array
print(type(a))                           # Prints "<class 'numpy.ndarray'>"
print(a.shape)                           # Prints "(3,)"
print(a[0], a[1], a[2])                  # Prints "1 2 3"
a[0] = 5                                 # Change an element of the array
print(a)                                 # Prints "[5, 2, 3]"
```

```
b = np.array([[1,2,3],[4,5,6]])          # Create a rank 2 array
print(b.shape)                           # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])         # Prints "1 2 4"
```

Using arange() function

import numpy as np ***a=np.arange(10)*** ***print(a)***	[0 1 2 3 4 5 6 7 8 9]
a=np.arange(11,20) ***print(a)***	[11 12 13 14 15 16 17 18 19]
a=np.arange(11,25,3) ***print(a)***	[11 14 17 20 23]

Numpy also provides many functions to create arrays:

a = np.zeros((2,2))	Create an array of all zeros having two rows and two columns
b = np.ones((1,2))	Create an array of all ones having one row and two columns
c = np.full((2,2), 7)	Create a constant array having two rows and two columns with constant value 7
d = np.eye(2)	Create a 2x2 identity matrix
e = np.random.random((2,2))	Create an array filled with random values
By default all these function have float data type	

How to change default data type

b=np.eye(2,3,dtype=np.int32) ***print(b)*** ***print("Data type of array b>>>>",b.dtype)***	***Output*** [[1 0 0] [0 1 0]] Data type of array b>> int32

```
import numpy as np
a = np.zeros((2,2))                      # Create an array of all zeros
print(a)                                 # Prints "[[ 0.  0.]
                                         #          [ 0.  0.]]"
```

b = np.ones((1,2)) ***print(b)***	# Create an array of all ones # Prints "[[1. 1.]]"
c = np.full((2,2), 7) ***print(c)***	# Create a constant array # Prints "[[7. 7.] # [7. 7.]]"
d = np.eye(2) ***print(d)***	# Create a 2x2 identity matrix # Prints "[[1. 0.] # [0. 1.]]"
b=np.eye(2,3) ***print(b)***	#prints [[1. 0. 0.] [0. 1. 0.]]
e = np.random.random((2,2)) ***print(e)***	# Create an array filled with random values # Might print "[[0.91940167 0.08143941] # [0.68744134 0.87236687]]"
np.random.randint(5, size=(3, 4))	*#creates random number between given range* #array([[0, 4, 0, 1], #[4, 1, 4, 2], #[3, 1, 2, 1]])

7.2 Shape and dimension of array

a.shape	Find the shape of the array
a.ndim	Find the dimension of the array

Example-2 One dimensional array

import numpy as np ***c=[1,4,5,1,6,7,8,9,12.5]*** ***a=np.array(c)*** ***print("Values of array", a)*** ***print("Data type of array>>>>",type(a))*** ***print("Shape of array>>>>",a.shape)*** ***print("Dimension of array>>>>",a.ndim)***	Values of array [1. 4. 5. 1. 6. 7. 8. 9. 12.5] Data type of array>>>> <class 'numpy.ndarray'> Shape of array>>>> (9,) Dimension of array>>>> 1

Example-3 Two dimensional array

import numpy as np c=[[1,4,5],[9,5,4]] a=np.array(c) print("Values of array", a) print("Data type of array>>>>",type(a)) print("Shape of array>>>>",a.shape) print("Dimension of array>>>>",a.ndim)	Values of array [[1 4 5] [9 5 4]] Data type of array>>>> <class 'numpy.ndarray'> Shape of array>>>> (2, 3) Dimension of array>>>> 2

Example-4 Two dimensional array

import numpy as np c=[[1,4,5],[9,5,4],[6,8,2],[8,1,2]] a=np.array(c) print("values of array", a) print("Data type of array>>>>",type(a)) print("Shape of array>>>>",a.shape) print("Dimension of array>>>>",a.ndim)	Values of array [[1 4 5] [9 5 4] [6 8 2] [8 1 2]] Data type of array>>>> <class 'numpy.ndarray'> Shape of array>>>> (4, 3) Dimension of array>>>> 2

7.3 Array indexing

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
import numpy as np
                        # Create the following rank 2 array with shape (3, 4)
                        # [[ 1  2  3  4]
                        #  [ 5  6  7  8]
                        #  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

              # Use slicing to pull out the subarray consisting of the first 2 rows
              # and columns 1 and 2; b is the following array of shape (2, 2):
              # [[2 3]
              #  [6 7]]
b = a[: 2, 1:3]

                       # A slice of an array is a view into the same data, so modifying it
                       # will modify the original array.
print(a[0, 1])         # Prints "2"
b[0, 0] = 77           # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])         # Prints "77"
```

Integer array indexing: When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

```
import numpy as np
a = np.array([[1,2], [3, 4], [5, 6]])
                                    # An example of integer array indexing.
                                   # The returned array will have shape (3,) and
```

```
print(a[[0, 1, 2], [0, 1, 0]])          # Prints "[1 4 5]"

                        # The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))          # Prints "[1 4 5]"

                        # When using integer array indexing, you can reuse the same
                        # element from the source array:
print(a[[0, 0], [1, 1]])          # Prints "[2 2]"

                        # Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))          # Prints "[2 2]"
```

Boolean array indexing: Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

```
import numpy as np
a = np.array([[1,2], [3, 4], [5, 6]])
bool_idx = (a > 2)          # Find the elements of a that are bigger than 2;
                            # this returns a numpy array of Booleans of the same
                            # shape as a, where each slot of bool_idx tells
                            # whether that element of a is > 2.
print(bool_idx)          # Prints "[[False False]
                            #          [ True  True]
                            #          [ True  True]]"
                        # We use boolean array indexing to construct a rank 1 array
                        # consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])          # Prints "[3 4 5 6]"
                            # We can do all of the above in a single concise statement:
print(a[a > 2])          # Prints "[3 4 5 6]"
```

7.4 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.

Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

```
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"

Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

NumPy - Arithmetic Operations

Input arrays for performing arithmetic operations such as add(), subtract(), multiply(), and divide() must be either of the same shape or should conform to array broadcasting rules.

Example

```
import numpy as np
a = np.arange(9, dtype = np.float_).reshape(3,3)

print('First array:')
print a
print ('\n')

print ('Second array:')
b = np.array([10,10,10])
print b
print('\n')

print('Add the two arrays:')
print np.add(a,b)
print('\n')

print('Subtract the two arrays:')
print np.subtract(a,b)
print('\n')

print('Multiply the two arrays:' )
print np.multiply(a,b)
print('\n')

print('Divide the two arrays:' )
print np.divide(a,b)
```

It will produce the following output −

```
First array:
[[ 0. 1. 2.]
 [ 3. 4. 5.]
 [ 6. 7. 8.]]

Second array:
[10 10 10]

Add the two arrays:
[[ 10. 11. 12.]
 [ 13. 14. 15.]
 [ 16. 17. 18.]]

Subtract the two arrays:
[[-10. -9. -8.]
 [ -7. -6. -5.]
 [ -4. -3. -2.]]

Multiply the two arrays:
[[ 0. 10. 20.]
 [ 30. 40. 50.]
 [ 60. 70. 80.]]

Divide the two arrays:
[[ 0. 0.1 0.2]
 [ 0.3 0.4 0.5]
 [ 0.6 0.7 0.8]]
```

Let us now discuss some of the other important arithmetic functions available in NumPy.

numpy.reciprocal()

This function returns the reciprocal of argument, element-wise. For elements with absolute values larger than 1, the result is always 0 because of the way in which Python handles integer division. For integer 0, an overflow warning is issued.

Example

import numpy as np a = np.array([0.25, 1.33, 1, 0, 100]) print('Our array is:') print(a) print('\n') print('After applying reciprocal function:') print np.reciprocal(a) print ('\n') b = np.array([100], dtype = int) print('The second array is:') print(b) print('\n') print('After applying reciprocal function:') print np.reciprocal(b)	It will produce the following output − Our array is: [0.25 1.33 1. 0. 100.] After applying reciprocal function: main.py:9: RuntimeWarning: divide by zero encountered in reciprocal print np.reciprocal(a) [4. 0.7518797 1. inf 0.01] The second array is: [100] After applying reciprocal function: [0]

numpy.power()

This function treats elements in the first input array as base and returns it raised to the power of the corresponding element in the second input array.

import numpy as np a = np.array([10,100,1000]) print('Our array is:') print(a) print(\n') print('Applying power function:') print np.power(a,2) print '\n') print('Second array:') b = np.array([1,2,3]) print(b) print ('\n')	It will produce the following output − Our array is: [10 100 1000] Applying power function: [100 10000 1000000] Second array: [1 2 3] Applying power function again: [10 10000 1000000000]

print('Applying power function again:') print np.power(a,b)	

numpy.mod()

This function returns the remainder of division of the corresponding elements in the input array. The function numpy.remainder() also produces the same result.

import numpy as np a = np.array([10,20,30]) b = np.array([3,5,7]) print('First array:') print(a) print('\n') print('Second array:') print(b) print('\n') print('Applying mod() function:') print np.mod(a,b) print('\n') print('Applying remainder() function:') print np.remainder(a,b)	It will produce the following output − First array: [10 20 30] Second array: [3 5 7] Applying mod() function: [1 0 2] Applying remainder() function: [1 0 2]

x = np.array([[1,2],[3,4]], dtype=np.float64) y = np.array([[5,6],[7,8]], dtype=np.float64) print(np.sqrt(x))	# Elementwise square root; produces the array # [[1. 1.41421356] # [1.73205081 2.]]

Note that unlike MATLAB, * is elementwise multiplication, not matrix multiplication. We instead use the dotfunction to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

```
import numpy as np
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])
v = np.array([9,10])
w = np.array([11, 12])
                                        # Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
                                # Matrix / vector product; both produce the rank 1 array [29 67]
```

```
print(x.dot(v))
print(np.dot(x, v))
                                # Matrix / matrix product; both produce the rank 2 array
                                # [[19 22]
                                #  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

```
import numpy as np
x = np.array([[1,2],[3,4]])
print(np.sum(x))                # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))        # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))        # Compute sum of each row; prints "[3 7]"
```

Summary

np.add(a,b)	Element wise addition of two matrix
np.subtract(a,b)	Element wise subtraction of two matrix
np.multiply(a,b)	Element wise multiplication of two matrix
np.divide(a,b)	Element wise division of two matrix
numpy.reciprocal()	This function returns the reciprocal of argument, element-wise.
numpy.power(a,b)	This function treats elements in the first input array as base and returns it raised to the power of the corresponding element in the second input array.
numpy.mod(a,b) np.remainder(a,b)	This function returns the remainder of division of the corresponding elements in the input array. The function numpy.remainder() also produces the same result.
np.sqrt(x))	Element wise square root of the matrix
v.dot(w)) np.dot(x, v))	Used to calculate inner product of two vectors or matrix multiplication.
np.sum(x))	Compute sum of all elements
np.sum(x, axis=0)	Compute sum of each column
np.sum(x, axis=1)	Compute sum of each row

7.5 NumPy - Iterating Over Array

NumPy package contains an iterator object **numpy.nditer**. It is an efficient multidimensional iterator object using which it is possible to iterate over an array. Each element of an array is visited using Python's standard Iterator interface.

Example-1

a=np.arange(12).reshape(3,4) print("Main array>>>", a) for row in a: print("Array after for loop>>",row)	Main array>>> [[0 1 2 3] [4 5 6 7] [8 9 10 11]] Array after for loop>> [0 1 2 3] Array after for loop>> [4 5 6 7] Array after for loop>> [8 9 10 11]

a=np.arange(12).reshape(3,4)
for row in a:
  for cell in row:
  print("Elements of array>>",cell)
	Elements of array>> 0
	Elements of array>> 1
	Elements of array>> 2
	Elements of array>> 3
	Elements of array>> 4
	Elements of array>> 5
	Elements of array>> 6
	Elements of array>> 7
	Elements of array>> 8
	Elements of array>> 9
	Elements of array>> 10
	Elements of array>> 11

Example-2

Going row by row and printing elements from each of the column (In order='C')

import numpy as np a = np.arange(0,60,5) a = a.reshape(3,4) print('Original array is:') print(a) print ('Modified array is:') for x in np.nditer(a, order='C'): print(x)	**Output** Original array is: [[0 5 10 15] [20 25 30 35] [40 45 50 55]] Modified array is: 0 5 10 15 20 25 30 35 40 45 50 55

Going column by column and printing elements from each of the row (In order='F')

import numpy as np a = np.arange(0,60,5) a = a.reshape(3,4) print('Original array is:') print(a) print ('Modified array is:') for x in np.nditer(a, order='F'): print(x)	**Output** Original array is: [[0 5 10 15] [20 25 30 35] [40 45 50 55]] Modified array is: 0 20 40 5 25 45 10 30 50 15 35 55

Modifying Array Values

The **nditer** object has another optional parameter called **op_flags**. Its default value is read-only, but can be set to read-write or write-only mode. This will enable modifying array elements using this iterator.

Example-3

import numpy as np a = np.arange(0,60,5) a = a.reshape(3,4) print('Original array is:') print(a) print('\n') for x in np.nditer(a, op_flags=['readwrite']): x[...]=2*x print('Modified array is:') print(a)	Its output is as follows − Original array is: [[0 5 10 15] [20 25 30 35] [40 45 50 55]] Modified array is: [[0 10 20 30] [40 50 60 70] [80 90 100 110]]

7.6 NumPy - Broadcasting

The term **broadcasting** refers to the ability of NumPy to treat arrays of different shapes during arithmetic operations. Arithmetic operations on arrays are usually done on corresponding elements. If two arrays are of exactly the same shape, then these operations are smoothly performed.

Example 1

import numpy as np a = np.array([1,2,3,4]) b = np.array([10,20,30,40]) c = a * b print(c)	Its output is as follows − [10 40 90 160]

If the dimensions of two arrays are dissimilar, element-to-element operations are not possible. However, operations on arrays of non-similar shapes is still possible in NumPy, because of the

broadcasting capability. The smaller array is **broadcast** to the size of the larger array so that they have compatible shapes.

Broadcasting is possible if the following rules are satisfied −
Array with smaller **ndim** than the other is prepended with '1' in its shape.
Size in each dimension of the output shape is maximum of the input sizes in that dimension.
An input can be used in calculation, if its size in a particular dimension matches the output size or its value is exactly 1.
If an input has a dimension size of 1, the first data entry in that dimension is used for all calculations along that dimension.
A set of arrays is said to be **broadcastable** if the above rules produce a valid result and one of the following is true −

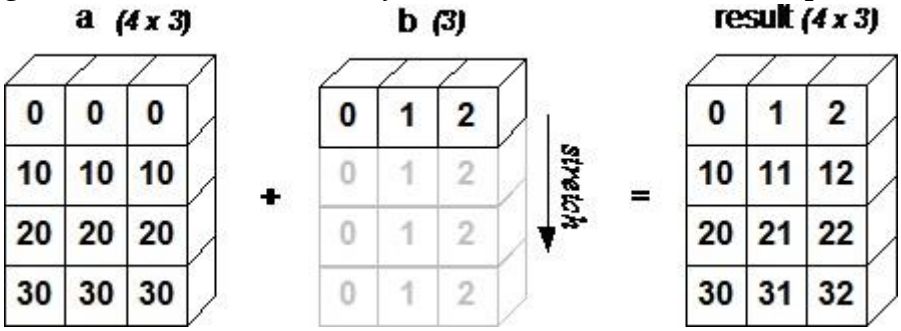
- Arrays have exactly the same shape.
- Arrays have the same number of dimensions and the length of each dimension is either a common length or 1.
- Array having too few dimensions can have its shape prepended with a dimension of length 1, so that the above stated property is true.

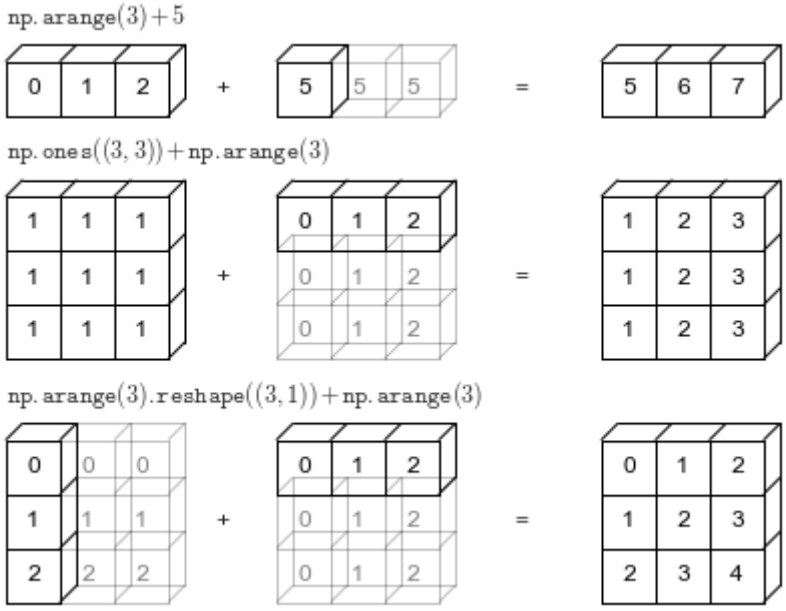
The following program shows an example of broadcasting.
Example 2

```
import numpy as np
a =np.array([[0.0,0.0,0.0],[10.0,10.0,10.0],[20.0,20.0,20.0],
[30.0,30.0,30.0]])
b = np.array([1.0,2.0,3.0])

print('First array:')
print(a)
print('\n')

print('Second array:' )
print(b)
print('\n')

print('First Array + Second Array')
print(a + b)
```

```
Output
First array:
[[ 0. 0. 0.]
 [ 10. 10. 10.]
 [ 20. 20. 20.]
 [ 30. 30. 30.]]

Second array:
[ 1. 2. 3.]

First Array + Second Array
[[ 1. 2. 3.]
 [ 11. 12. 13.]
 [ 21. 22. 23.]
 [ 31. 32. 33.]]
```

The following figure demonstrates how array **b** is broadcast to become compatible with **a**.





The light boxes represent the broadcasted values: again, this extra memory is not actually allocated in the course of the operation, but it can be useful conceptually to imagine that it is.

8 Python Pandas

Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures. The name Pandas is derived from the word Panel Data – an Econometrics from Multidimensional data.
In 2008, developer Wes McKinney started developing pandas when in need of high performance, flexible tool for analysis of data.

Prior to Pandas, Python was majorly used for data munging and preparation. It had very little contribution towards data analysis. Pandas solved this problem. Using Pandas, we can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data — load, prepare, manipulate, model, and analyze.

Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.

Key Features of Pandas

- Fast and efficient DataFrame object with default and customized indexing.
- Tools for loading data into in-memory data objects from different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of date sets.
- Label-based slicing, indexing and subsetting of large data sets.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High performance merging and joining of data.
- Time Series functionality.

8.1 How to read excel file

```
import pandas as pd
a=pd.read_excel("C:/Users/Kuldeep/Desktop/Files/data.xlsx")
print(a)
```

Output

```
   Roll no       Name  Class  Marks
0     1120   Anukampa     11     70
1     1121  Harsimrat     12     60
2     1122     Muskan     11     80
3     1123    Sukhbir     10     85
4     1124   Niharika      9     78
```

Reading data from specific sheet

```
a=pd.read_excel("C:/Users/Kuldeep/Desktop/Files/data.xlsx", 'employee')
print(a)
```

```
  Empid    Ename  Salary
0    e1  rupleen   40000
1    e2  akansha   35000
2    e3    dolly   38000
3    e4    anmol   45000
4    e5    gagan   35000
```

File without header

```
a=pd.read_excel("C:/Users/Kuldeep/Desktop/Files/data.xlsx", 'employee', header=None)
print(a)
```

```
    0         1      2
0  e1  rupleen  40000
1  e2  akansha  35000
2  e3    dolly  38000
3  e4    anmol  45000
4  e5    gagan  35000
```

```
a=pd.read_excel("C:/Users/Kuldeep/Desktop/Files/data.xlsx", 'employee',
header=None,names=("Empid","name","salary","city"))
print(a)
```

```
  Empid     name  salary        city
0    e1  rupleen   40000     Patiala
1    e2  akansha   35000     Patiala
2    e3    dolly   38000  Chandigarh
3    e4    anmol   45000  Chandigarh
4    e5    gagan   35000     Patiala
```

Selecting column

a["name"]	0 rupleen 1 akansha 2 dolly 3 anmol 4 gagan
a[["name","city"]]	

	name	city
0	rupleen	Patiala
1	akansha	Patiala
2	dolly	Chandigarh
3	anmol	Chandigarh
4	gagan	Patiala

a.loc[2,'salary'] #accessing single cell	38000

Slicing

a[: 2]

	Empid	name	salary	city
0	e1	rupleen	40000	Patiala
1	e2	akansha	35000	Patiala

a[: : 2]

	Empid	name	salary	city
0	e1	rupleen	40000	Patiala
2	e3	dolly	38000	Chandigarh
4	e5	gagan	35000	Patiala

a[: : -1]

	Empid	name	salary	city
4	e5	gagan	35000	Patiala
3	e4	anmol	45000	Chandigarh
2	e3	dolly	38000	Chandigarh
1	e2	akansha	35000	Patiala
0	e1	rupleen	40000	Patiala

a.loc[1:3]
#selection by index

	Empid	name	salary	city
1	e2	akansha	35000	Patiala
2	e3	dolly	38000	Chandigarh
3	e4	anmol	45000	Chandigarh

Getting maximum value from column **a["col3"].max()**	**Output** **4500**
a["col3"].min()	**3500**
a["salary"].mean()	38600.0

Printing employee name who belong to patiala

a["name"][a["city"]=="Patiala"]	0 rupleen 1 akansha 4 gagan

Writing data to excel sheet

Writing with sheet name
a.to_excel("C:/Users/Kuldeep/Desktop/Files/newfile1.xlsx", sheet_name="newemployee")

newemployee.xlsx

	Empid	Ename	Salary
0	e1	rupleen	40000
1	e2	akansha	35000
2	e3	dolly	38000
3	e4	anmol	45000
4	e5	gagan	35000

newemployee

Writing without index
a.to_excel("C:/Users/Kuldeep/Desktop/Files/newfile.xlsx", sheet_name="newstudent", index=False)

newstudent.xlsx

Roll no	Name	Class	Marks
1120	Anukampa	11	70
1121	Harsimrat	12	60
1122	Muskan	11	80
1123	Sukhbir	10	85
1124	Niharika	9	78

newstudent

Converting data to array

import numpy as np b=np.array(a) print(b) print(b[: ,1]) #accessing 2nd column	**Output** [['e1' 'rupleen' 40000] ['e2' 'akansha' 35000] ['e3' 'dolly' 38000] ['e4' 'anmol' 45000] ['e5' 'gagan' 35000]] 2nd column... ['rupleen' 'akansha' 'dolly' 'anmol' 'gagan']
print("2nd row...",b[1 ,:])	2nd row... ['e2' 'akansha' 35000]

8.2 How to read CSV file

```
import pandas as pd
a=pd.read_csv("C:/Users/Kuldeep/Desktop/Files/data.csv")
print(a)
```

Output

```
  Empid    Ename  Salary
0    e1  rupleen   40000
1    e2  akansha   35000
2    e3    dolly   38000
3    e4    anmol   45000
4    e5    gagan   35000
```

9 Reading and Writing to text files in Python

Python provides inbuilt functions for creating, writing and reading files. There are two types of files that can be handled in python, normal text files and binary files (written in binary language,0s and 1s).
Text files: In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in python by default.

Binary files: In this type of file, there is no terminator for a line and the data is stored after converting it into machine understandable binary language.
In this article, we will be focusing on opening, closing, reading and writing data in a text file.

File Access Modes

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once its opened. These modes also define the location of the **File Handle** in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python.

Read Only ('r') : Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises I/O error. This is also the default mode in which file is opened.

Read and Write ('r+') : Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exists.

Write Only ('w') : Open the file for writing. For existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exists.

Write and Read ('w+') : Open the file for reading and writing. For existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.

Append Only ('a') : Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

Append and Read ('a+') : Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

9.1 Opening a File

It is done using the open() function. No module is required to be imported for this function.
File_object = open(r"File_Name","Access_Mode")
The file should exist in the same directory as the python program file else, full address of the file should be written on place of filename.

Note: The **r** is placed before filename to prevent the characters in filename string to be treated as

special character. For example, if there is \temp in the file address, then \t is treated as the tab character and error is raised of invalid address. The r makes the string raw, that is, it tells that the string is without any special characters. The r can be ignored if the file is in same directory and address is not being placed.

```
# Open function to open the file "MyFile1.txt"
# (same directory) in append mode and
file1 = open("MyFile.txt","a")

# store its reference in the variable file1
# and "MyFile2.txt" in D:\Text in file2
file2 = open(r"D:\Text\MyFile2.txt","w+")
```

Here, file1 is created as object for MyFile1 and file2 as object for MyFile2

9.2 Closing a file

close() function closes the file and frees the memory space acquired by that file. It is used at the time when the file is no longer needed or if it is to be opened in a different file mode.

```
File_object.close()
# Opening and Closing a file "MyFile.txt"
# for object name file1.
file1 = open("MyFile.txt","a")
file1.close()
```

9.3 Writing to a file

There are two ways to write in a file.

write() : Inserts the string str1 in a single line in the text file.

```
File_object.write(str1)
```

writelines() : For a list of string elements, each string is inserted in the text file.Used to insert multiple strings at a single time.

```
File_object.writelines(L) for L = [str1, str2, str3]
```

Reading from a file

There are three ways to read data from a text file.

read() : Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.

```
File_object.read([n])
```

readline() : Reads a line of the file and returns in form of a string.For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.

```
 File_object.readline([n])
```

readlines() : Reads all the lines and return them as each line a string element in a list.

```
  File_object.readlines()
```

Note: '\n' is treated as a special character of two bytes

```
# Program to show various ways to read and
# write data in a file.
file1 = open("myfile.txt","w")
L = ["This is Delhi \n","This is Paris \n","This is London \n"]

# \n is placed to indicate EOL (End of Line)
file1.write("Hello \n")
file1.writelines(L)
file1.close() #to change file access modes

file1 = open("myfile.txt","r+")

print "Output of Read function is "
print file1.read()
print

# seek(n) takes the file handle to the nth
# bite from the beginning.
file1.seek(0)

print "Output of Readline function is "
print file1.readline()
print

file1.seek(0)

# To show difference between read and readline
print "Output of Read(9) function is "
print file1.read(9)
print

file1.seek(0)

print "Output of Readline(9) function is "
print file1.readline(9)

file1.seek(0)
# readlines function
print "Output of Readlines function is "
print file1.readlines()
print
file1.close()
```

Output:

Output of Read function is

```
Hello
This is Delhi
This is Paris
This is London


Output of Readline function is
Hello


Output of Read(9) function is
Hello
Th

Output of Readline(9) function is
Hello

Output of Readlines function is
['Hello \n', 'This is Delhi \n', 'This is Paris \n', 'This is London \n']
```

10 Python - Regular Expressions

A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The module **re** provides full support for Perl-like regular expressions in Python. The re module raises the exception re.error if an error occurs while compiling or using a regular expression.

We would cover two important functions, which would be used to handle regular expressions. But a small thing first: There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as **r'expression'**.

The most common uses of regular expressions are:

- re.match()
- re.search()
- re.findall()
- re.split()
- re.sub()

re.match(*pattern*, *string*):

This method finds match if it occurs at start of the string. For example, calling match() on the string 'AV Analytics AV' and looking for a pattern 'AV' will match. However, if we look for only Analytics, the pattern will not match. Let's perform it in python now.

```
import re
line='AV Analytics Vidhya AV'
result = re.match(r'AV', line)
print(result)
print("Matching string...",result.group())
```

Output

```
<_sre.SRE_Match object; span=(0, 2), match='AV'>
Matching string... AV
```

Above, it shows that pattern match has been found. To print the matching string we'll use method group (It helps to return the matching string). Use "r" at the start of the pattern string, it designates a python raw string.

Let's now find 'Analytics' in the given string. Here we see that string is not starting with 'AV' so it should return no match. Let's see what we get:

```
import re
line='AV Analytics Vidhya AV'
result = re.match(r'Analytics', line)
print(result)
```

Output

```
None
```

Flag I for case insensitive

```
line='AV Analytics Vidhya AV'
result = re.match(r'av', line,re.I)                    #flag re.I for case insensitive
print(result)
print("Matching string...",result.group())
```

Output

```
AV
```

There are methods like start() and end() to know the start and end position of matching pattern in the string.

```
import re
line='AV Analytics Vidhya AV'
result = re.match(r'AV', line)
print(result.start())
print(result.end())
```

Output:

```
0
2
```

re.search(*pattern*, *string*):
It is similar to match() but it doesn't restrict us to find matches at the beginning of the string only. Unlike previous method, here searching for pattern 'Analytics' will return a match.

```
result = re.search(r'Analytics', 'AV Analytics Vidhya AV')
print(result.group(0))
```

Output:
Analytics

Here you can see that, search() method is able to find a pattern from any position of the string but it only returns the first occurrence of the search pattern.

Groups

```
line='Today is wednesday and tomorrow is thursday'
result = re.search(r'(.*)and(.*)', line,re.I) #flag re.I for case insenstive
print(result)
print("Matching string...",result.group())
print("Group-1...",result.group(1))
print("Group-2...",result.group(2))
print("All groups in tuple...",result.groups())
```

Output:
Matching string... Today is wednesday and tomorrow is thursday
Group-1... Today is wednesday
Group-2... tomorrow is thursday
All groups in tuple... ('Today is wednesday ', ' tomorrow is thursday')

In [ ]

re.findall (*pattern*, *string*):
It helps to get a list of all matching patterns. It has no constraints of searching from start or end. If we will use method findall to search 'AV' in given string it will return both occurrence of AV. While searching a string, I would recommend you to use **re.findall()** always, it can work like re.search() and re.match() both.

```
result = re.findall(r'AV', 'AV Analytics Vidhya AV')
print(result)
```

Output:
['AV', 'AV']

re.split(*pattern*, *string*, [*maxsplit=0*]):
This methods helps to split *string* by the occurrences of given *pattern*.

```
result=re.split(r'y','Analytics Vidhya')
print(result)
```

Output:
['Anal', 'tics Vidh', 'a']

Above, we have split the string "Analytics" by "y". Method split() has another argument "**maxsplit**". It has default value of zero. In this case it does the maximum splits that can be done, but if we give value to maxsplit, it will split the string. Let's look at the example below:

```
result=re.split(r'y','Analytics Vidhya',maxsplit=1)
result
```

```
['Anal', 'tics Vidhya']
```

re.sub(*pattern*, *repl*, *string*):
It helps to search a pattern and replace with a new sub string. If the pattern is not found, *string* is returned unchanged.

```
result=re.sub(r'India','the World','AV is largest Analytics community of India')
print(result)
```

Output:
'AV is largest Analytics community of the World'

What are the most commonly used operators?
Regular expressions can specify patterns, not just fixed characters. Here are the most commonly used operators that helps to generate an expression to represent required characters in a string or file. It is commonly used in web scrapping and text mining to extract required information.

Operators	Description
.	Matches with any single character except newline '\n'.
?	match 0 or 1 occurrence of the pattern to its left
+	1 or more occurrences of the pattern to its left
*	0 or more occurrences of the pattern to its left
\w	Matches with a alphanumeric character whereas \W (upper case W) matches non alphanumeric character.
\d	Matches with digits [0-9] and /D (upper case D) matches with non-digits.
\s	Matches with a single white space character (space, newline, return, tab, form) and \S (upper case S) matches any non-white space character.
\b	boundary between word and non-word and /B is opposite of /b
[..]	Matches any single character in a square bracket and [^..] matches any single character not in square bracket
\	It is used for special meaning characters like \. to match a period or \+ for plus sign.
^ and $	^ and $ match the start or end of the string respectively

{n,m}	Matches at least n and at most m occurrences of preceding expression if we write it as {,m} then it will return at least any minimum occurrence to max m preceding expression.
a| b	Matches either a or b
( )	Groups regular expressions and returns matched text
\t, \n, \r	Matches tab, newline, return

Some Examples of Regular Expressions

Problem 1: Return the first word of a given string

Solution-1 Extract each character **(**using "**.**"**)**

```
import re
result=re.findall(r'.','AV is largest Analytics community of India')
print(result)
```

Output:

```
['A', 'V', ' ', 'i', 's', ' ', 'l', 'a', 'r', 'g', 'e', 's', 't', ' ', 'A', 'n', 'a', 'l', 'y', 't', 'i', 'c', 's', ' ', 'c', 'o', 'm', 'm', 'u', 'n', 'i', 't', 'y', ' ', 'o', 'f', ' ', 'I', 'n', 'd', 'i', 'a']
```

Above, space is also extracted, now to avoid it use "**\w**" instead of "**.**".

```
import re
result=re.findall(r'\w','AV is largest Analytics community of India')
print(result)
```

Output:

```
['A', 'V', 'i', 's', 'l', 'a', 'r', 'g', 'e', 's', 't', 'A', 'n', 'a', 'l', 'y', 't', 'i', 'c', 's', 'c', 'o', 'm', 'm', 'u', 'n', 'i', 't', 'y', 'o', 'f', 'I', 'n', 'd', 'i', 'a']
```

Solution-2 Extract each word **(**using "*****" or "**+**"**)**

```
result=re.findall(r'\w*','AV is largest Analytics community of India')
print(result)
```

Output:

```
['AV', '', 'is', '', 'largest', '', 'Analytics', '', 'community', '', 'of', '', 'India', '']
```

Again, it is returning space as a word because "*****" returns zero or more matches of pattern to its left. Now to remove spaces we will go with "**+**".

```
result=re.findall(r'\w+','AV is largest Analytics community of India')
print(result)
```

Output:

```
['AV', 'is', 'largest', 'Analytics', 'community', 'of', 'India']
```

Solution-3 Extract each word **(**using "^"**)**

```
result=re.findall(r'^\w+','AV is largest Analytics community of India')
print(result)
```

Output:
['AV']

If we will use "$" instead of "^", it will return the word from the end of the string. Let's look at it.

```
result=re.findall(r'\w+$','AV is largest Analytics community of India')
print result
```

Output:
['India']

Problem 2: Return the first two character of each word

Solution-1 Extract consecutive two characters of each word, excluding spaces (using "**\w**"**)**

```
result=re.findall(r'\w\w','AV is largest Analytics community of India')
print(result)
```

Output:
['AV', 'is', 'la', 'rg', 'es', 'An', 'al', 'yt', 'ic', 'co', 'mm', 'un', 'it', 'of', 'In', 'di']

Solution-2 Extract consecutive two characters those available at start of word boundary (using "**\b**"**)**

```
result=re.findall(r'\b\w.','AV is largest Analytics community of India')
print(result)
```

Output:
['AV', 'is', 'la', 'An', 'co', 'of', 'In']