Object Oriented Programming System (OOPs)

What is Class?

- In Python, everything is an object.
- To create objects we required some Model or Plan or Blue print, which is nothing but **class**.
- We can write a class to represent **properties** (attributes) and **actions** (behavior) of object.
- **Properties** can be represented by **variables**
- **Actions** can be represented by **Methods**.

Class = Variables + Methods

Defining a class:

- We can define a class by using class keyword.

  Syntax:

```
class ClassName:
    ''' documentation string '''
    variables
    methods
```

Documentation string:

- It represents description of the class.
- Within the class doc string is always optional.
- We can get doc string by using the following 2 ways.

  1. print(ClassName.__doc__)
  2. help(ClassName)

Example:

```
class Student:
    ''' This is student class with required data'''

print(Student.__doc__)
help(Student)
```

```
D:\PythonApps>python test.py
 This is student class with required data
Help on class Student in module __main__:

class Student(builtins.object)
 |  This is student class with required data
 |
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Example for class:

```
class Student:
    '''Developed by Adiseshu for python'''
    def __init__(self):
        self.name='Sai'
        self.age=25
        self.marks=80

    def talk(self):
        print("Hello I am :",self.name)
        print("My Age is:",self.age)
        print("My Marks are:",self.marks)
```

What is Object?

- Physical existence of a class is nothing but object.
- We can create any number of objects for a class.

Syntax to create object:

referencevariable = ClassName()

Example:

s = Student()

What is Reference Variable?

- The variable which can be used to refer object is called reference variable.
- By using reference variable, we can access **properties and methods** of object.

Write a Python program to create a Student class and Creates an object to it.
Call the method talk() to display student details

```
class Student:
    '''Developed by Adiseshu for python'''

    def __init__(self,name,age,marks):
        self.name=name
        self.age=age
        self.marks=marks

    def talk(self):
        print("Hello I am :",self.name)
        print("My Age is:",self.age)
        print("My Marks are:",self.marks)

s1=Student("Sai",30,90)
s1.talk()

s2=Student("Ram",40,90)
s1.talk()
```

```
D:\Python Workspace>python test.py
Hello I am : Sai
My Age is: 30
My Marks are: 90
Hello I am : Sai
My Age is: 30
My Marks are: 90
```

What is ‘ self ’ ?

- self is the default variable which is always pointing to current object.
  (Just like **this** keyword in Java, C# and Java Script)
- It is used to access **instance variables** and **instance methods** of object.

Note:

1. self should be first parameter inside **constructor**
   def __init__(self):

2. self should be first parameter inside **instance methods**
   def talk(self):

Working with Constructor

- Constructor is a special method in python.
- The name of the constructor in python is fixed and it must be **__init__(self)**
- Constructor will be executed automatically at the time of object creation.
- The main purpose of constructor is to declare and initialize **instance variables**.
- For each and every object, constructor will be executed only once.
- Constructor must take at least one argument that is **self.** Otherwise error.
- Constructor is optional and if we are not providing any constructor then python will provide **default constructor.**

Example:

```
def __init__(self,name,rollno,marks):
  self.name=name
  self.rollno=rollno
  self.marks=marks
```

Eg: Program to demonstrate constructor will execute only once per object:

```
class Test:
    def __init__(self):
        print("Constructor exeuction...")

    def m1(self):
        print("Method execution...")
t1=Test()
t2=Test()
t3=Test()
t1.m1()
```

```
D:\PythonApps>python test.py
Constructor exeuction...
Constructor exeuction...
Constructor exeuction...
Method execution...
```

Eg: Complete Program:

```
class Student:
    ''' This is student class with required data'''

    def __init__(self,x,y,z):
        self.name=x
        self.rollno=y
        self.marks=z

    def display(self):
        print("Student Name:{}\nRollno:{} \nMarks:{}".format(self.name,self.rollno,se
lf.marks))
```

```
s1=Student("Sai",101,80)
s1.display()
s2=Student("ram",102,100)
s2.display()
```

```
D:\PythonApps>python test.py
Student Name:Sai
Rollno:101
Marks:80
Student Name:ram
Rollno:102
Marks:100
```

Differences between Methods and Constructors:

Method	Constructor
➢ Name of method can be any name	➢ Constructor name should be always **__init__**
➢ Method will be executed if we call that method	➢ Constructor will be executed automatically at the time of object creation.
➢ Per object, method can be called any number of times.	➢ Per object, Constructor will be executed only once
➢ Inside method we can write **business logic**	➢ Inside Constructor we have to **declare and initialize instance variables**

Reuse members of one class inside another class

We can use members of one class inside another class by using the following ways

1. **By Composition (Has-A Relationship)**
2. **By Inheritance (IS-A Relationship)**

By Composition (Has-A Relationship):

- By using Class Name or by creating object we can access members of one class inside another class is nothing but composition (Has-A Relationship).

- The main advantage of Has-A Relationship is **Code Reusability**.

Eg :

```
class Car:
    def __init__(self,name,model,color):
        self.name=name
        self.model=model
        self.color=color

    def getinfo(self):
        print("Car Name:{} , Model:{} and Color:{}".format(self.name,self.model,self.color))

class Employee:
    def __init__(self,ename,eno,car):
        self.ename=ename
        self.eno=eno
        self.car=car

    def empinfo(self):
        print("Employee Name:",self.ename)
        print("Employee Number:",self.eno)
        print("Employee Car Info:",end=" ")
        self.car.getinfo()

c=Car("Innova","2.5V","Grey")
e=Employee('Sai',10000,c)
e.empinfo()
```

```
D:\PythonApps>python test.py
Employee Name: Sai
Employee Number: 10000
Employee Car Info: Car Name:Innova , Model:2.5V and Color:Grey
```

By Inheritance (IS-A Relationship)

- Whatever **variables, methods and constructors** available in the **parent** class by default available to the child classes and we are not required to rewrite. Hence the main advantage of inheritance is Code Reusability and we can extend existing functionality with some more extra functionality.

Syntax :

class ParentClassName:
    Members

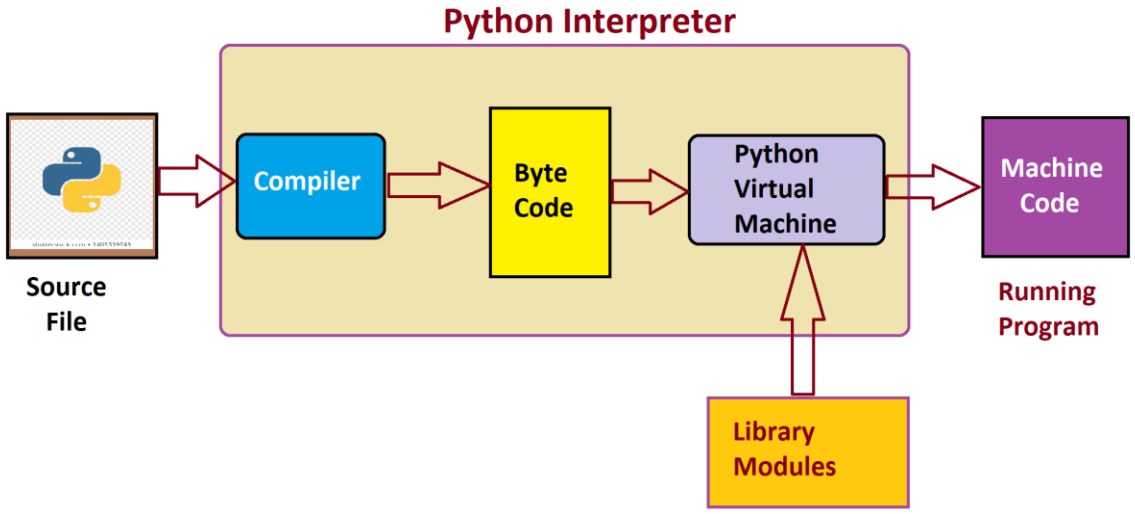
class ChildClassName (**ParentClassName**):
    Parent Members
    Child Members

Eg:

```
class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age

    def eatndrink(self):
        print('Eat and Drink')

class Developer(Person):
    def __init__(self,name,age,eno,esal):
        super().__init__(name,age)
        self.eno=eno
        self.esal=esal

    def code(self):
        print("Busy with Coding in Python")

    def developerInfo(self):
        print("Name:",self.name)
        print("Age:",self.age)
        print("Number:",self.eno)
        print("Salary:",self.esal)

d=Developer('Ram', 48, 100, 10000)
d.eatndrink()
d.code()
d.developerInfo()
```

Note: super() is a built-in method which is useful to call the
super class constructors,
variables and
methods from the **child class**.

Python Internal Flow



Compiler:

- Compiler is a program that convert source code (.py file) into Byte Code.

Byte Code:

- The bytecode is a **low-level platform-independent** representation of your source code.
- In Python, the bytecode is stored in a **.pyc** file and these are stored in a folder named **__pycache__.**
- This folder is automatically created when you try to **import** another file that you created.

PVM:

- After compilation, the bytecode is sent for execution to the PVM.
- The PVM is an **interpreter** that is responsible to run the bytecode to Machine Code.
- Generally, PVM is specific to the target machine.
- The default implementation of PVM is **CPython** which is written in the **C programming language**.

Example:

mymath.py

```
def double(num):
    print(num+num)
```

first.py

```
import mymath

print(mymath.double(10))
```

>python first.py