ADVANCE PYTHON

MOHAN REDDY

TABLE OF CONTENTS

ADVANCE PYTHON

OBJECT ORIENTED PROGRAMMING (OOP'S) :

- Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic.
- Object-oriented programming (OOP) is a style of programming characterized by the identification of classes of objects closely linked with the methods (functions) with which they are associated. It also includes ideas of inheritance of attributes and methods.
- The main advantages of Object-oriented programming is:
  - Security
  - Reusability
  - Application enhancement
- In Object-oriented programming mainly we need to discuss about class and object
- **Class:** is a collection of variables and methods.
- **Object :** is an instance of class or it is used to represent a class.
- Apart from class an object if any programming language wants to become as object oriented then that programming language must and should satisfy the following features.

FEATURES OF OBJECT-ORIENTED PROGRAMMING:

1. **Encapsulation**
2. **Abstraction**
3. **Polymorphism**
4. **Inheritance**

ENCAPSULATION:

- It is the process of providing restriction to access the variables and methods.
- Why we need encapsulation means to prevent the data from modification.
- We can achieve the encapsulation by using private variables and methods.

ABSTRACTION:

- It is the process of hiding the implementation but providing service
- Abstraction is the concept of object-oriented programming that "shows" only essential attributes and "hides" unnecessary information.
- The main purpose of abstraction is hiding the unnecessary details from the users.
- For example, when we are driving a car, we are only concerned about driving the car like start/stop the car, accelerate/ break, etc.
- We are not concerned about how the actual start/stop mechanism or accelerate/brake process works internally. We are just not interested in those details.

POLYMORPHISM:

- Poly means many and morph means form or behavior.
- Polymorphism is one of the core concepts of object-oriented programming (OOP) and describes situations in which something occurs in several different forms.
- In polymorphism, operator or method will show different behavior every time when we pass different data types of arguments and no of arguments.
- **Types of polymorphism:**
    1. Static polymorphism
    2. Dynamic polymorphism
- Static polymorphism is related to method overloading.
- Dynamic polymorphism is related to method overriding.

INHERITANCE:

- Inheritance is the process of creating new class from existing class.
- In inheritance process, existing class treated as base class or parent class or super class.
- In inheritance process, new class treated as derived class or child class or sub class.
- The main advantage of inheritance is code reusability and extensibility.
- In inheritance process, child class will get all features of its parent class.

TYPES OF INHERITANCE:

1. **Single** :
   - It is the process of creating new class from single base class
2. **Multi-level** :
   - It is the process of creating new class from already derived class
3. **Hierarchical** :
   - It is the process of creating multiple child classes from single base class
4. **Multiple** :
   - It is the process of creating new class from two or more base classes
5. **Hybrid** :
   - It is the combination of multilevel, multiple, hierarchical inheritance.

HOW TO CREATE A CLASS:

Syntax: Class Classname

Ex: Class Student

HOW TO CREATE OBJECT TO THE CLASS:

Syntax: Objectreferencevariable = Classname( )

Ex: s = Student( )

- The variable which is used to refer an object is called object reference variable.

Program to create a class and get the details of class:

```
class Student:
    '''This is student class for display
student details'''

#help(Student)
s=Student()
print(s.__doc__)
```

Program to create Student class and get the details of student:

```
class Student:

    def __init__(self):
        self.sid=123
        self.sname="sai"
        self.saddress="hyderabad"


    def display(self):
        print("Student id is:",self.sid)
        print("Student name is:",self.sname)
        print("Student address is:",self.saddress)


s1=Student()
s2=Student()
s1.display()
s2.display()
```

CONSTRUCTOR:

- Constructor is a special method of class.
- Constructor is used to initialize the data to the variables.
- Constructor name should be __init__(self):
- Self is the first parameter of the constructor.
- Self is used to access current class members.
- Constructor will execute automatically when we create object to class.
- We can create any no of objects to the class, for every object constructor will execute once.

METHOD:

- Method is a reusable piece of code.
- Method is used to implement any business logic
- Method name can be of any name.
- Self is the first parameter of the method.
- Method will execute only when we call that method
- For every object we can call a method any no of times
- Inside the class Including constructor is optional
- Inside the class Including method is optional

Ex with parameterized constructor:

```
class Student:
    def __init__(self,x,y,z):
        self.sid=x
        self.sname=y
        self.saddress=z

    def display(self):
        print("Student id is:",self.sid)
        print("Student name is:",self.sname)
        print("Student address
is:",self.saddress)

s1=Student(101,"sai","hyderabad")
s2=Student(102,"mohan","sr nagar")
s1.display()
s2.display()
```

Ex to display student details without constructor:

```
class Student:
    def getdata(self):
        self.sid=int(input("Enter sid:"))
        self.sname=input("Enter sname:")
        self.saddress=input("Enter saddress:")

    def display(self):
        print("Student id is:",self.sid)
        print("Student name is:",self.sname)
        print("Student address is:",self.saddress)

s1=Student()
s1.getdata()
s1.display()
```

Ex to display student details without Methods:

```
class Student:
    def __init__(self,x,y,z):
        self.sid=x
        self.sname=y
        self.saddress=z


s1=Student(101,"sai","hyderabad")
s2=Student(102,"mohan","sr nagar")
print(s1.__dict__)
print(s2.__dict__)
```

TYPES OF VARIABLES:

- We can use 3 types of variables in python class
  1. Instance variables (object level)
  2. Static variables (Class level)
  3. Local variables (Method level)

INSTANCE VARIABLES:

- If the value of a variable is varied from object to object, then such type of variables are called instance variables.
- For every object a separate copy of instance variables will be created.

- **Where we can declare Instance variables:**
  - Inside Constructor by using self
  - Inside Instance method by using self
  - Outside of the class by using object reference variable

Ex to declare instance variables

```
class Test:
    def __init__(self):
        self.a=10

    def m1(self):
        self.b=20

t=Test()
t.m1()
t.c=30
print(t.__dict__)
```

How to access instance variables:

- We can access instance variables within the class by using self.

  Ex: self.variablename

- We can access instance variables outside of the class by using object reference variable.

  Ex: objectreference.varablename

Ex to access instance variables

```
class Test:
    def __init__(self):
        self.a=10
        self.b=20

    def m1(self):
        print("within the class")
        print(self.a)
        print(self.b)

t=Test()
t.m1()
print("outside of the class")
print(t.a)
print(t.b)
```

Note: We can create any no of object to the class , for every object a separate copy of instance variables will be create ,if we change or delete one copy of instance variables then other copy of instance variables will not effect.

How to delete instance variables:

- We can delete instance variables within the class as follows.

  Ex: del self.variablename

- We can delete instance variables outside of the class as follows.

  Ex: del objectreference.varablename

Ex to delete or change instance variables

```
class Test:
    def __init__(self):
        self.a=10
        self.b=20
        self.c=30

    def m1(self):
        del self.a

t1=Test()
t2=Test()
print(t1.__dict__)
print(t2.__dict__)
t1.m1()
del t2.b
t1.c=44
print(t1.__dict__)
print(t2.__dict__)
```

STATIC VARIABLES:

- If the value of variable is not varied from object to object, then such type of variables, we can declare directly inside the class but outside of the methods is called static variables.
- We can access static variables either by using class name or object reference but recommended is to use class name.
- For all objects only one copy of static variables will be create if we change or delete static variables it will affect all objects.

Ex with static variables

```
class Test:
    a=10 #static variable

    def __init__(self):
        self.b=20 #instance variable

t1=Test()
t2=Test()
print("t1:",t1.a,t1.b)
print("t2:",t2.a,t2.b)
Test.a=99
t1.b=45
print("t1:",t1.a,t1.b)
print("t2:",t2.a,t2.b)
```

LOCAL VARIABLES:

- The variables which are declare inside the particular method and that variables are available for only that method is called local variables.
- Local variables cannot access outside of the methods.
- Local variables can only be access within the same method where we declare it.
- Local variables are also called as temporary variables.
- Once method execution starts these variables will be create
- Once method execution completes these will be destroyed
- Local variables can be declared inside the method without using self.

Ex with Local variables

```
class Test:
    def m1(self):
        a=10
        print(a)
        #print(b)#NameError: name 'b' is not
defined

    def m2(self):
        b=20
        print(b)
        #print(a)#NameError: name 'a' is not
defined

t=Test()
t.m1()
t.m2()
```

TYPES OF METHODS:

- We can use 3 types of methods in python class
    1. Instance methods
    2. Class methods
    3. Static methods

INSTANCE METHODS:

- Inside the method implementation when we use instance variables then such type of methods are called instance methods.
- While declaring instance methods we should pass self
- self is the first parameter of instance methods
- We can access instance methods within the class by using self
  Ex: self.methodname ( )
- We can access instance methods outside of the class by using object reference
  Ex: objectreference.methodname( )

CLASS METHODS:

- Class methods are very rarely used methods in python
- Inside the method implementation when we use only class or static variables then such type of methods are called class methods.
- To make the method as class method then we use @classmethod decorator
- While declaring class methods we should pass class variable i.e. cls
- Using cls we can access class or static variables inside the class methods
- We can access class methods either by using class name or object reference

STATIC METHODS:

- These methods are general utility methods, as per programmer requirement he/she can implement any type of logic is called static methods.
- While declaring static methods we should not pass any self or cls.
- To make a method as static then we use @staticmethod decorator
- We can access static methods either by using class name or object reference

Ex:

```
class Student:
    inst="durgasoft" #static variable
    def __init__(self,m1,m2,m3):
        self.m1=m1
        self.m2=m2
        self.m3=m3
        #self.avg()

    def avg(self): #instance method
        #return (self.m1+self.m2+self.m3)/3
        print((self.m1+self.m2+self.m3)/3)

    @classmethod
    def m1(cls):
        print(cls.inst)

    @staticmethod
    def add(a,b):
        print("sum is:",a+b)

    @staticmethod
    def f1():
        print("Hello")


s1=Student(78,87,69)
s2=Student(90,86,94)
#print(s1.avg())
#print(s2.avg())
s1.avg()
s2.avg()
Student.m1()
s1.add(10,20)
s1.f1()
```

Where we can access static variables:

- Inside the constructor by using self or class name
- Inside the instance methods by using self or class name
- Inside the class methods by using cls or class name
- Inside the static methods by using class name
- Outside of the class by using class name or object reference

Ex:

```
class Test:
    a=10 #static variable

    def __init__(self):
        print("Inside the constructor")
        print(self.a)
        print(Test.a)

    def m1(self):
        print("Inside the instance method")
        print(self.a)
        print(Test.a)

    @classmethod
    def m2(cls):
        print("Inside the class method")
        print(cls.a)
        print(Test.a)

    @staticmethod
    def m3():
        print("Inside the static method")
        print(Test.a)

t=Test()
t.m1()
t.m2()
t.m3()
print("Outside of the class")
print(Test.a)
print(t.a)
```

Inner class:

- Sometimes we can declare a class inside another class is called inner class.
- A class which is having one more class within it is called inner class.
- Without existing one type of object if there is no chance of existing another type of object then we use inner class mechanism.

Ex: Class Car:
    Code…
    Class Engine:
        Code….

Ex: Class University:
    Code…
    Class College:
        Code….

Ex with Inner class:

```
class Outer:
    def __init__(self):
        print("Outer class constructor")

    def f1(self):
        print("outer class method")

    class Inner:
        def __init__(self):
            print("Inner class constructor")

        def m1(self):
            print("Inner class method")

'''o=Outer()
i=o.Inner()
i.m1()'''

'''i=Outer().Inner()
i.m1()'''

#Outer().Inner().m1()

o=Outer()
o.f1()
i=o.Inner()
i.m1()
```

GC (GARBAGE COLLECTOR):

- In old languages like C++, programmer is responsible for both creation and destruction of objects.
- Usually programmer taking very much care while creating object, but neglecting destruction of useless objects.
- Because of his neglect, total memory can be filled with useless objects which create memory problems and total application will be down with Out of memory error.
- But in Python, We have some assistant which is always running in the background to destroy useless objects, because this assistant the chance of failing Python program with memory problems is very less. This assistant is nothing but Garbage Collector.
- Hence the main objective of Garbage Collector is to destroy useless objects.
- If an object does not have any reference variable then that object eligible for Garbage Collection.
- By default Garbage collector is enabled, but we can disable based on our requirement. In this context we can use the following functions of gc module.
    - gc .isenabled()
        - Returns True if GC enabled
    - gc.disable()
        - To disable GC explicitly
    - gc.enable()
        - To enable GC explicitly

Ex:

```
import gc

print(gc.isenabled())
gc.disable()
print(gc.isenabled())
gc.enable()
print(gc.isenabled())
```

DESTRUCTORS:

- Destructor is a special method and the name should be __del__ Just before destroying an object Garbage Collector always calls destructor to perform clean-up activities (resource de allocation activities like close database connection etc.).
- Once destructor execution completed then Garbage Collector automatically destroys that object.
- **Note:** The job of destructor is not to destroy object and it is just to perform clean-up activities.

Ex:

```
import time

class Test:
    def __init__(self):
        print("Constructor execution")

    def __del__(self):
        print("Destructor execution")

t=Test()
time.sleep(5)
print("End of the program")
```

How to find the number of references of an object:

- To find no of references of an object then we use getrefcount() function of sys module.
- sys.getrefcount(object reference)
  Note: For every object, Python internally maintains one default reference variable self.

Ex:

```
import sys

class Test:
    pass

t1 = Test()
t2 = t1
t3 = t1
t4 = t1
print(sys.getrefcount(t1))
```

ENCAPSULATION:

- Encapsulation is the process of providing restriction to access variables and methods.
- Why we need encapsulation means to prevent the data from modification.
- We can achieve the encapsulation mechanism by using private variables and private methods.
- **Private :** Access within the same class
- **Protected:** Access within the same class and also in its child class
- **Public:** Access within the class and outside of the class.
- In python private variables and methods can be declare starts with double underscore.
  Ex: __empid or def __m1(self)
- In python protected variables and methods can be declare starts with single underscore.
  Ex: _empid or def _m1(self)
- In python public variables and methods can be declare starts with no underscore.
  Ex: empid or def m1(self)

Ex:

```
class Test:
    x=10 #public
    _y=20 #protected
    __z=30 #private

    def __init__(self):
        print("within the class")
        print(self.x)
        print(self._y)
        print(self.__z)

t=Test()
print("outside of the class")
print(t.x)
print(t._y)
print(t.__z) #AttributeError: 'Test' object
has no attribute '__z'
```

Ex:

```
class Parent:
    x=10 #public
    _y=20#protected
    __z=30 #private

class Child(Parent):
    print(Parent.x)
    print(Parent._y)
    #print(Parent.__z) #AttributeError: type
object 'Parent' has no attribute '_Child__z'

c=Child()
```

Ex:

```
class Car:
    def __init__(self):
        self.__updatesoftware()

    def __updatesoftware(self): #private
method
        print("Updating software")

c=Car()
#c.__updatesoftware() #AttributeError: 'Car'
object has no attribute '__updatesoftware'
```

Ex:

```
class Car:
    __name=""
    __maxspeed=0

    def __init__(self):
        self.__name="verna"
        self.__maxspeed=100
        print(self.__name)
        print(self.__maxspeed)

    def drive(self):
        self.__maxspeed=200
        print("Driving")
        print(self.__name)
        print(self.__maxspeed)

c=Car()
#c.__maxspeed=200 #maxspeed will not change
c.drive()
```

INHERITANCE:

- Inheritance is a powerful feature in object oriented programming.
- Inheritance is the process of creating new class from existing class.
- In inheritance process, existing class will be treated as base class or parent class or super class.
- In inheritance process, new class will be treated as derived class or child class or sub class.
- In inheritance process always child class will get all features from its parent class.
- The main advantage of Inheritance is reusability and extensibility.

Syntax :

```
class Baseclass:
        body of the base class
class Derivedclass (Baseclass):
        body of the derived class
```

Ex with single and multi-level inheritance:

```
class Branch:

    def getbranchdata(self):
        self.bcode=int(input("Enter branch code:"))
        self.bname=input("Enter branch name:")
        self.baddress=input("Enter branch address:")


    def displaybranchdata(self):
        print("Branch code is:",self.bcode)
        print("Branch name is:",self.bname)
        print("Branch address is:",self.baddress)
```

```
class Employee(Branch):

    def getempdata(self):
        self.eid=int(input("Ente eid:"))
        self.ename=input("Enter ename:")
        self.eaddress=input("Enter eaddress:")

    def displayempdata(self):
        print("Empid is:",self.eid)
        print("Ename is:",self.ename)
        print("Eaddress is:",self.eaddress)


class Salary(Employee):

    def getsal(self):
        self.basic=int(input("Enter basic salary:"))

    def calculate(self):
        self.DA=self.basic*0.06
        self.HRA=self.basic*0.4
        self.Gross=self.basic+self.DA+self.HRA

    def displaysal(self):
        print("Basic is:",self.basic)
        print("DA is:",self.DA)
        print("HRA is:",self.HRA)
        print("Gross is:",self.Gross)


s=Salary()
s.getbranchdata()
s.getempdata()
s.getsal()
s.calculate()
s.displaybranchdata()
s.displayempdata()
s.displaysal()
```

Ex with hierarchical inheritance:

```
class Parent:
    def f1(self):
        print("Parent class method")

class Child1(Parent):
    def f2(self):
        print("Child1 class method")

class Child2(Parent):
    def f3(self):
        print("Child2 class method")

c1=Child1()
c2=Child2()
c1.f1()
c1.f2()
c2.f1()
c2.f3()
```

isinstance():

- It is used to check whether the object is an instance of particular class or not

issubclass():

- It is used to check whether the class is a subclass of particular class or not

Ex:

```
class Parent:
    def f1(self):
        print("Parent class method")

class Child1(Parent):
    def f2(self):
        print("Child1 clas method")
```

```
class Child2(Parent):
    def f3(self):
        print("child2 class method")

c1=Child1()
c2=Child2()

print(isinstance(c1,Child1)) #true
print(isinstance(c2,Child2)) #true
print(isinstance(c1,Child2)) #false
print(isinstance(c2,Child1)) #false
print(isinstance(c1,Parent))# true
print(isinstance(c2,Parent))# true

print(issubclass(Child1,Parent)) #true
print(issubclass(Child2,Parent)) #true
print(issubclass(Child1,Child2)) #false
print(issubclass(Child2,Child1)) #false
```

Ex with Multiple inheritance:

- In Python multiple inheritance implemented by using MRO technique (Method Resolution Order)
- It is the order in which a method is searched for in a class hierarchy and is especially useful in python because python supports multiple inheritance.
- In Python, the MRO is from bottom to top and left to right

```
class A:
    def f1(self):
        print("F1 function of class A")

class B:
    def f2(self):
        print("F2 function of class B")

class C(A,B):
    def f3(self):
        print("F3 function of class C")
```

```
c=C()
c.f1()
c.f2()
c.f3()
```

Ex :

```
class A:
    def f1(self):
        print("F1 function of class A")

class B:
    def f1(self):
        print("F1 function of class B")

class C(A,B):
    def f3(self):
        B.f1(self)
        print("F3 function of class C")

c=C()
c.f1()
c.f3()
```

Ex with Hierarchical inheritance:

```
class A:
    def f1(self):
        print("F1 function of class A")

class B:
    def f1(self):
        print("F1 function of class B")

class C(A,B):
    def f3(self):
        B.f1(self)
        print("F3 function of class C")
```

```
class D(C):
    def f4(self):
        print("F4 function of class D")

class E(C):
    def f5(self):
        print("F5 function of class E")

e=E()
e.f1()
e.f3()
e.f5()

d=D()
d.f1()
d.f3()
d.f4()
```

POLYMORPHISM:

- Poly means many and morph means behavior or forms.
- If we take a function, it will show different behavior when we change the data types of arguments or no of arguments.
- Polymorphism can be of two types.
  1. Static or compile time
  2. Dynamic or runtime
- Static polymorphism is a refinement technique and it is related to overloading.
- Dynamic polymorphism is a replacement technique and it is related to overriding.
- **Overloading :**
  - Operator overloading
  - Method overloading
  - Constructor overloading
- **Overriding :**
  - Method overriding
  - Constructor overriding

OPERATOR OVERLOADING:

- We can use the same operator for multiple purposes, which is nothing but operator overloading.
- Python supports operator overloading.
- Ex : + operator can be used for Arithmetic addition and String concatenation

  print(10+20) #30

  print('durga'+'soft') #durgasoft
- Ex : * operator can be used for multiplication and string repetition purpose.

  print(10*20) #200

  print('durga'*3) #durgadurgadurga

program to use + operator for our class objects:

```
class Book:
    def __init__(self,pages):
        self.pages=pages

b1=Book(10)
b2=Book(20)
print(b1+b2) #TypeError: unsupported operand type(s) for +: 'Book' and 'Book'
```

- For every operator Magic Methods are available. To overload any operator we have to override that Method in our class.
- Internally + operator is implemented by using __add__( ) method.
- This method is called magic method for + operator.
- We have to override this method in our class.

Ex:

```
class Book:
    def __init__(self,pages):
        self.pages=pages

    def __add__(self, other):
        return self.pages+other.pages

    def __mul__(self, other):
        return self.pages*other.pages

b1=Book(10)
b2=Book(20)
print(b1+b2)
print(b1*b2)
```

Ex:

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def __mul__(self, other):
        return self.salary * other.days

class TimeSheet:
    def __init__(self, name, days):
        self.name = name
        self.days = days

e = Employee('Durga', 500)
t = TimeSheet('Durga', 25)
print('This Month Salary:', e * t)
```

METHOD OVERLOADING:

- If 2 methods having same name but different type of arguments then those methods are said to be overloaded methods.
- Ex: m1(int a)
  m1(double d)
- But in Python Method overloading is not possible.
- If we are trying to declare multiple methods with same name and different number of arguments then Python will always consider only last method.

Ex:

```
class Test:
    def m1(self):
        print("no arg method")

    def m1(self,a):
        print("one arg method")

    def m1(self,a,b):
        print("two arg method")

t=Test()
#t.m1()
#t.m1(10)
t.m1(10,20)
```

CONSTRUCTOR OVERLOADING:

- If 2 constructors having same name but different type of arguments then those methods are said to be overloaded constructors.
- But in Python constructor overloading is not possible.
- If we are trying to declare multiple constructors with same name and different number of arguments then python will always consider only last constructor.

Ex:

```
class Test:
    def __init__(self):
        print("no arg constructor")

    def __init__(self,a):
        print("one arg constructor")

    def __init__(self,a,b):
        print("two arg constructor")

#t =Test()
#t =Test(10)
t=Test(10,20)
```

Method and Constructor overriding:

- Whatever members available in the parent class those are by default available to the child class through inheritance.
- If the child class not satisfied with parent class implementation then child class is allowed to redefine that method in the child class based on its requirement. This concept is called overriding.
- Overriding concept applicable for both methods and constructors.
- From overriding method of child class, we can call parent class method also by using super() method.

Ex:

```
class Parent:
    def property(self):
        print("Cash+Gold+Lands")

    def car(self):
        print("Alto car")

class Child(Parent):
    def car(self):
        super().car()
        print("Benz car")

c=Child()
c.property()
c.car()
```

Ex :

```
class Parent:
    def __init__(self):
        print("Parent class constructor")

class Child(Parent):
    def __init__(self):
        super().__init__()
        print("Child class constructor")

c=Child()
```

ABSTRACT METHOD AND ABSTRACT CLASS:

ABSTRACT METHOD:

- A method which does not contain any implementation, which contains only declaration, is called abstract method.
- Sometimes as programmer we know how to declare a method but we don't know how to implement it then we use abstract method.
- Abstract method should be override.
- To create a method as abstract then we use @abstractmethod decorator.
- To leave abstract method with no implementation then we use pass keyword.

  Ex:

  ```
  @abstractmethod
  def m1(self):
      pass
  ```

ABSTRACT CLASS:

- A class which contains one or more abstract methods is called abstract class.
- Abstract class can also contain non abstract methods.
- To make a class as abstract then that class should inherit or extends from predefined abstract base class.
- In python ABC is an Abstract Base Class.
- ABC and abstract method decorator are present in abc module
- Abstract class cannot instantiate directly so that we need to create or derive new class from abstract class to provide functionality to its abstract functions.
- We cannot create object for abstract class.
- Partial implementation of the class is called abstract class.
- How many no of abstract methods present in abstract class, all methods must be implemented in its child classes.

Ex :

```
from abc import ABC,abstractmethod
class Test(ABC):
    @abstractmethod
    def m1(self):
        pass
```

Ex :

```
from abc import ABC,abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def wheels(self):
        pass

    def engine(self):
        print("Bs6 engine")

    @abstractmethod
    def color(self):
        pass
```

```
class Car(Vehicle):
    def wheels(self):
        print("Car:4 wheels")

    def color(self):
        print("Car:color is red")

class Bike(Vehicle):
    def wheels(self):
        print("Bike:2 wheels")

    def color(self):
        print("Bike:color is black")

c=Car()
c.wheels()
c.engine()
c.color()

b=Bike()
b.wheels()
b.engine()
b.color()
```

EXCEPTION HANDLING:

- In general, when we type python code and execute there may be a chance of 2 types of error occurrence.
  1. Syntactical errors
  2. Runtime errors

SYNTACTICAL ERRORS:

- These errors occur when we miss : or () or "" or writing wrong spelling to the syntax
- These errors do not cause any harmful to the program execution
- These errors can be identify and rectify by the programmer at the time of writing code.

RUNTIME ERRORS:

- Runtime errors is nothing but exception
- Exceptions will cause abnormal termination of program execution.
- These errors cannot be identify and rectify by the programmer.
- The errors which occur at the time of program execution due to the invalid inputs are called runtime errors.
- To handle the exceptions we use exception handling mechanism.

EXCEPTION HANDLING MECHANISM:

1. Logical implementation
2. Try except implementation

- First priority we should give to logical implementation
- If any exceptions not possible to handle by using logical implementation then we should go for try except implementation
- The unexpected or unwanted event will disturb the normal flow of program execution is called exception.
- **Note:** exception handling mechanism is only for runtime errors not for syntactical errors.

- **Note:** exception handling mechanism is not the process of repairing errors it is just the process to show alternative way of program execution.
- In python programming every exception is a class

Built-in exception classes:

1. Name error
2. Value error
3. Type error
4. Zero division error
5. File not found error
6. File exist error etc.

- These all are subclass of super class exception
- **Exception** is the super class of all sub class exceptions.
- Whenever an exception occurs PVM will create the corresponding exception object and will check for handling code.
- If handling code is not available then Python interpreter terminates the program abnormally and prints corresponding exception information to the console. The rest of the program won't be executed.

Ex with syntax error:

```
print("hello"    #SyntaxError: unexpected EOF
while parsing

def f1()    #SyntaxError: invalid syntax
  print("Hello")
```

Ex with runtime error:

```
a=10
print(A) #NameError: name 'A' is not defined

a=int(input("Enter a number:")) #Enter a
number:abc
                                    #ValueError:
invalid literal for int() with base 10: 'abc'
print(a)

print(10+"abc") #TypeError: unsupported
operand type(s) for +: 'int' and 'str'
print(10/0) #ZeroDivisionError: division by
zero
```

Ex with logical implementation:

```
a=int(input("Enter Num1:"))
b=int(input("Enter Num2:"))
print("result is:",a/b)
```

Note: In the above example when user give second number as zero then we will get ZeroDivisionError, we can handle this by using logical statements as follows.

Ex:

```
a=int(input("Enter Num1:"))
b=int(input("Enter Num2:"))
if b==0:
    print("second num cant be zero")
else:
    print("result is:",a/b)
```

Note: All the exceptions may not possible to handle using logical statements in that case we use try except implementation.

Working with try except implementation:

Syntax:

try:

Statements

except:

Statements

- In general inside the try block we include risky statements and inside except block we include handling statements.
- If no exception occurs inside the try block then try block statements will execute and except block will ignore.
- If any exception occurs inside the try block then try block will ignore and except block will execute.

Ex:

```
try:
    a = int(input("Enter Num1:"))
    b = int(input("Enter Num2:"))
    print("result is:", a / b)
except :
    print("something went wrong")
```

- In the above example, any type of exception raised then we will get same message that is something went wrong.
- To display the proper exception information then we use specific except block.

TYPES OF EXCEPT BLOCKS:

- **Default except block :** except block without any exception class
- **Specific except block :** except block with exception class

Ex:

```
try:
    a = int(input("Enter Num1:"))
    b = int(input("Enter Num2:"))
    print("result is:", a / b)
except ZeroDivisionError as msg:
    print(msg)
```

Note: we can also include multiple except blocks.

Ex:

```
try:
    a = int(input("Enter Num1:"))
    b = int(input("Enter Num2:"))
    print("result is:", a / b)
except ZeroDivisionError as msg:
    print(msg)
except ValueError as msg:
    print(msg)
```

Note: instead of multiple except blocks, we can use single except block with multiple exception classes.

Ex:

```
try:
    a = int(input("Enter Num1:"))
    b = int(input("Enter Num2:"))
    print("result is:", a / b)
except(ZeroDivisionError,ValueError)as msg:
    print(msg)
```

Note: instead of multiple exception classes, we can use super class of exceptions.

Ex:

```
try:
    a = int(input("Enter Num1:"))
    b = int(input("Enter Num2:"))
    print("result is:", a / b)
except Exception as msg:
    print(msg)
```

Note: While working with specific except block and default except block make sure that default except block should be last otherwise we will get syntax error.

Ex:

```
try:
    a = int(input("Enter Num1:"))
    b = int(input("Enter Num2:"))
    print("result is:", a / b)
except:
    print("something went wrong")
except ZeroDivisionError as msg:
    print(msg)
#SyntaxError: default 'except:' must be last
```

Ex:

```
try:
    a = int(input("Enter Num1:"))
    b = int(input("Enter Num2:"))
    print("result is:", a / b)
except ZeroDivisionError as msg:
    print(msg)
except:
    print("something went wrong")
```

Case study of try except blocks:

```
try:
  statement1
  statement2
  statement3
except:
   statement4
statement5
```

Ex:

```
try:
  print("statement-1")
  #print("statement-2")
  print(10/0)
  print("statement-3")
except :
    print(10/0)
   #print("statement-4")
print("statement-5")
```

- **Case1**: if there is no exception then 1,2,3,5 and it is normal termination.

- **Case2**: if there is an exception at statement-2 then 1, 4, 5 and it is normal termination.

- **Case3**: if there is an exception at statement-2 but corresponding except block is not matched then 1, and it is abnormal termination.

- **Case4**: if there is an exception at statement-4 then it is always abnormal termination.

Working with finally block:

- Including finally block is optional.
- It is not recommended to maintain clean up code (Resource DE allocating Code or Resource Releasing code) inside try block because there is no guarantee for the execution of every statement inside try block always.

- It is not recommended to maintain clean up code inside except block, because if there is no exception then except block won't be executed.
- Hence we required some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether exception handled or not handled. Such type of best place is nothing but finally block.
- Hence the main purpose of finally block is to maintain clean up code or for executing mandatory statements.

Ex:

```
try:
    #print(10/0)
    print("try block")
except:
    print("except block")
finally:
    print("finally block")
```

- There is only one situation where finally block won't be executed i.e. whenever we are using os._exit (0) function.
- Whenever we are using os._exit(0) function then Python Virtual Machine itself will be shutdown. In this particular case finally won't be executed.
- Here 0 represents status code and it indicates normal termination

Ex:

```
import os
try:
    #print(10/0)
    print("try block")
    #os._exit(0)
except:
    print("Except block")
    os._exit(0)
finally:
    print("finally block")
```

Nested try-except-finally blocks:

- We can take try-except-finally blocks inside try or except or finally blocks. i.e. nesting of try except-finally is possible.

Ex:

```
try:
    #print(10/0)
    print("Outer try")
    try:
        print(10/0)
        print("Inner try")
    except ValueError:
        print("Inner except")
    finally:
        print("Inner finally")
except:
    print("Outer except")
finally:
    print("Outer finally")
```

Note: in general if exception occurs inside the inner try block,
Inner except block should take responsible to handle the exception.
If inner except block not handle the exception then outer except block will Handle that exception.

Important points related to try except finally:

- If the control entered into try block then compulsory finally block will be executed. If the control not entered into try block then finally block won't be executed.
- Whenever we are writing try block, compulsory we should write except or finally block.i.e without except or finally block we cannot write try block.
- Whenever we are writing except block, compulsory we should write try block. i.e. except without try is always invalid.
- Whenever we are writing finally block, compulsory we should write try block. i.e. finally without try is always invalid.

- We can write multiple except blocks for the same try, but we cannot write multiple finally blocks for the same try

TYPES OF EXCEPTIONS:

1. Predefined exceptions
2. User defined exceptions

- **Predefined exceptions** also called as built-in exceptions.
- The exceptions which are raised automatically by Python virtual machine whenever a particular event occurs are called pre-defined exceptions.

- **User defined exceptions** are also called as customized exceptions.
- Sometimes we have to define and raise exceptions explicitly to indicate that something goes wrong ,such type of exceptions are called User Defined Exceptions or Customized Exceptions
- Programmer is responsible to define these exceptions and Python not having any idea about these. Hence we have to raise explicitly based on our requirement by using "**raise"** keyword.

Working with user defined exceptions:

Syntax:

```
class classname(predefined exception class name):
    def __init__(self,arg):
        self.msg=arg
```

Ex:

```
class Too_oldException(Exception):
    def __init__(self,arg):
        self.msg=arg

class Too_youngException(Exception):
    def __init__(self,arg):
        self.msg=arg

age=int(input("Enter your age:"))
if age>60:
    raise Too_oldException("Age should not
exceed 60")
elif age<16:
    raise Too_youngException("Age should not
be under 16")
else:
    print("You are eligible to take policy")
```

Ex:

```
class Too_oldException(Exception):
    def __init__(self,arg):
        self.msg=arg

class Too_youngException(Exception):
    def __init__(self,arg):
        self.msg=arg
try:
    age = int(input("Enter your age:"))
    if age > 60:
        raise Too_oldException("Age should not
exceed 60")
```

```
    elif age < 16:
        raise Too_youngException("Age should
not be under 16")
    else:
        print("You are eligible to take
policy")
#except
(Too_youngException,Too_oldException,ValueErro
r) as msg:
except Exception as msg:
    print(msg)
```

FILE HANDLING:

- As the part of programming requirement, we have to store our data permanently for future purpose. For this requirement we should go for files.
- Files are very common permanent storage areas to store our data.

TYPES OF FILES:

- There are 2 types of files
  1. Text Files:
     - Usually we can use text files to store character data
       Ex: abc.txt
  2. Binary Files:
     - Usually we can use binary files to store binary data like images, video files, audio files etc...

OPENING A FILE:

- Before performing any operation (like read or write) on the file, first we have to open that file. For this we should use Python's inbuilt function open()
- But at the time of open, we have to specify mode, which represents the purpose of opening file.
  Ex: f = open (filename, mode)

FILE MODES:

- **r** ------- Open an existing file for read operation. The file pointer is positioned at the beginning of the file. If the specified file does not exist then we will get FileNotFoundError.This is default mode.

- **w** ------ Open an existing file for write operation. If the file already contains some data then it will be overridden. If the specified file is not already available then this mode will create that file.

- **a** -------- Open an existing file for append operation. It won't override existing data. If the specified file is not already avaialble then this mode will create a new file.

- **r+** ------- To read and write data into the file. The previous data in the file will not be deleted. The file pointer is placed at the beginning of the file.

- **w+** ------- To write and read data. It will override existing data.

- **a+** ------- To append and read data from the file. It won't override existing data.

- **x** -------- To open a file in exclusive creation mode for write operation. If the file already exists then we will get FileExistsError.

- **Note:** All the above modes are applicable for text files. If the above modes suffixed with 'b' then these represents for binary files.
  Ex: rb, wb, ab, r+b, w+b, a+b, xb

CLOSING A FILE:

- After completing our operations on the file, it is highly recommended to close the file. For this we have to use close () function.
  Ex : f.close()

PROPERTIES OF FILE OBJECT:

- Once we open a file and we got file object, we can get various details related to that file by using its properties.
- **name** --- Name of opened file
- **mode** ---Mode in which the file is opened
- **closed** ---Returns boolean value indicates that file is closed or not
- **readable()---** Returns boolean value indicates that whether file is readable or not
- **writable()---** Returns boolean value indicates that whether file is writable or not.
-

Ex:

```
f=open("sample.txt","w")
print("file name is:",f.name)
print("file mode is:",f.mode)
print("is file readable?",f.readable())
print("is file writable?",f.writable())
print("is file closed?",f.closed)
f.close()
print("is file closed?",f.closed)
```

Writing data to text files:

- We can write character data to the text files by using the following 2 methods.
    - write(str)
    - writelines(list of lines)

Ex:

```
f=open("abc.txt",'w')
f.write("Hello\n")
f.write("Durgasoft\n")
f.write("Hyderabad\n")
print("Data written to the file")
f.close()
```

Ex:

```
f=open("sample.txt",'w')
l=["sai\n","ram\n","mohan\n","durga\n"]
f.writelines(l)
print("List data written to the file")
f.close()
```

- **Note:** In the above program, data present in the file will be overridden every time if we run the program. Instead of overriding if we want append operation then we should open the file with append mode.

Ex:

```
f=open("xyz.txt",'a')
f.write("Hello durgasoft\n")
print("data written to the file")
f.close()
```

- **Note:** while writing data by using write () methods, compulsory we have to provide line seperator (\n), otherwise total data should be written to a single line.

Reading Character Data from text files:

- We can read character data from text file by using the following read methods.

  - read() --- To read total data from the file
  - read(n) --- To read 'n' characters from the file
  - readline()---To read only one line
  - readlines()---To read all lines into a list

Ex:

```
try:
    f = open("sample.txt", 'r')
    #data = f.read()
    #data=f.read(5)
    #data=f.readline()
    data=f.readlines()
    print(data)
    f.close()
except FileNotFoundError as msg:
    print(msg)
```

The with statement:

- The with statement can be used while opening a file.
- We can use this to group file operation statements within a block.
- The advantage of with statement is it will take care closing of file, after completing all operations automatically even in the case of exceptions also, and we are not required to close explicitly.

Ex:

```
with open("abcd.txt","w") as f:
    f.write("Hello\n")
    f.write("Hyderabad\n")
    print("is file closed?",f.closed)
print("is file closed?",f.closed)
```

The seek() and tell() methods:

- **tell():** We can use tell() method to return current position of the cursor (file pointer) from beginning of the file.

- **Seek() :** We can use seek() method to move cursor(file pointer) to specified location.

Ex:

```
f=open("abc.txt",'r')
print(f.tell())
print(f.read(3))
print(f.tell())
f.seek(11)
print(f.tell())
```

Ex to modify the file:

```
data="hello hyderabad"
f=open("rrr.txt",'w')
f.write(data)
```

```
with open("rrr.txt","r+") as f:
    print("current cursor position:",f.tell())
    text=f.read()
    print(text)
    print("current cursor position:",
f.tell())
    f.seek(6)
    print("current cursor position:",
f.tell())
    f.write("Durgasoft")
    print("current cusor position:",f.tell())
    f.seek(0)
    text=f.read()
    print("Data after modification")
    print(text)
```

How to check a particular file exists or not?

- We can use os library to get information about files in our computer.
- os module has path sub module, which contains isfile() function to check whether a particular file exists or not.
  Ex: os.path.isfile(file name)

Ex:

```
import os

fname=input("Enter file name to cehck:")

if os.path.isfile(fname):
    print("file is exist:",fname)
    f=open(fname,'r')
else:
    print("file does not exist:",fname)
    os._exit(0)
text=f.read()
print("content of the file")
print(text)
```

Ex to print the number of lines, words and characters present in the given file

```
import os
fname=input("Enter file name to cehck:")
if os.path.isfile(fname):
    print("file is exist:",fname)
    f=open(fname,'r')
else:
    print("file does not exist:",fname)
    os._exit(0)

lcount=ccount=wcount=0
for line in f:
    lcount=lcount+1
    ccount=ccount+len(line)
    words=line.split(" ")
    wcount=wcount+len(words)
print("No of lines:",lcount)
print("No of characters:",ccount)
print("No of words:",wcount)
```

ZIPPING AND UNZIPPING FILES:

- It is very common requirement to zip and unzip files.
- The main advantages are:
  - To improve memory utilization
  - We can reduce transport time
  - We can improve performance.
- To perform zip and unzip operations, Python contains one in-built module zip file. This module contains a class : ZipFile

To create Zip file:

- We have to create ZipFile class object with name of the zip file, mode and constant ZIP_DEFLATED. This constant represents we are creating zip file.

  Ex : f = ZipFile("files.zip","w","ZIP_DEFLATED")

- Once we create ZipFile object, we can add files by using write () method.

  Ex: f.write (filename)

Ex:

```
from zipfile import *

f=ZipFile("files.zip","w",ZIP_DEFLATED)
f.write("xyz.txt")
f.write("abc.txt")
f.write("rrr.txt")
print("Zip file created")
f.close()
```

To perform unzip operation:

- We have to create ZipFile object as follows

  Ex: f = ZipFile ("files.zip","r",ZIP_STORED)

- ZIP_STORED represents unzip operation. This is default value and hence we are not required to specify.
- Once we created ZipFile object for unzip operation, we can get all file names present in that zip file by using namelist () method.

  Ex: names = f.namelist()

Ex:

```
from zipfile import *

f=ZipFile("files.zip",'r',ZIP_STORED)
names=f.namelist()
print(names)
print(type(names))

for name in names:
    print("file name is:",name)
    f=open(name,"r")
    text=f.read()
    print("content of the file")
    print(text)
```

WORKING WITH CSV FILES:

- CSV stands for Comma Separated Values.
- CSV is a simple file format used to store tabular data, such as a spread sheet or database.
- A CSV file stores tabular data (numbers and text) in plain text. Each line of the file is a data record.
- Each record consists of one or more fields, separated by commas.
- The use of the comma as a field separator is the source of the name for this file format.
- For working CSV files in python, there is an inbuilt module called csv.

Ex to write csv files:

```
import csv

with open("emp.csv","w",newline='')as f:
    w=csv.writer(f)
    w.writerow(["Eno","Ename","Esal","Eaddr"])
    n=int(input("Enter no of employees:"))
    for i in range(n):
        eno=input("Enter emp no:")
        ename=input("Enter emp name:")
        esal=input("Enter emp sal:")
        eaddr=input("Enter emp address:")
        w.writerow([eno,ename,esal,eaddr])
print("Employee data written to csv file")
```

Ex to read csv files:

```
import csv
f=open("emp.csv","r")
r=csv.reader(f)
data=list(r)

for line in data:
    for word in line:
        print(word,"\t",end="")
    print()
```

PICKLING AND UNPICKLING:

- Pickling is the process of writing state of object into a file
- Unpickling is the process of reading state of object from file
- We can implement pickling and unpickling by using pickle module
- pickle module contains dump() function to perform pickling
  Ex: pickle.dump()
- pickle module contains load() function to perform unpickling
  Ex: pickle.load()

Ex:

```
import pickle

class Employee:
    def __init__(self,eid,ename,eaddress):
        self.eid=eid
        self.ename=ename
        self.eaddress=eaddress

    def display(self):

print(self.eid,self.ename,self.eaddress)

with open("emp.dat","wb") as f:
    e=Employee(101,"sai","hyderabad")
    pickle.dump(e,f)
    print("pickling of employee object
completed")

with open("emp.dat","rb") as f:
    obj=pickle.load(f)
    print("Display emp information after
unpickling")
    obj.display()
```

- A DAT file is a data file that contains specific information about the program used to create it.
- This file always has the . dat file extension, which is a generic format that can contain any information – video, audio, PDF, and virtually any other type of file.

REGULAR EXPRESSIONS:

- A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.
- RegEx can be used to check if a string contains the specified search pattern.
- Python has a built-in package called re, which can be used to work with Regular Expressions.
- If we want to represent a group of Strings according to a particular format/pattern then we should go for Regular Expressions.
- Regular Expressions is a declarative mechanism to represent a group of Strings according to particular format/pattern.
- The main important application areas of Regular Expressions are :

1. To develop validation frameworks/validation logic
2. To develop Pattern matching applications.

- re module contains several inbuilt functions to use Regular Expressions very easily in our applications.

Function of re module:

1. compile()
2. finditer()
3. match()
4. fullmatch()
5. search()
6. findall()
7. sub()
8. subn()
9. split()

- **compile () :** This function is used to compile a pattern into Regex Object.
- **finditer():** Returns an Iterator object which , Match object for every Match
- On Match object we can call the following methods.
    1. start() : Returns start index of the match
    2. end() : Returns end+1 index of the match
    3. group() : Returns the matched string

Ex:

```
import re
count=0
pattern=re.compile("ab")
match=pattern.finditer("abaababa")
for m in match:
  count+=1

print(m.start(),"...",m.end(),"...",m.group())
print("The number of occurrences: ",count)
```

CHARACTER CLASSES:

- We can use character classes to search a group of characters
  - [abc]===>Either a or b or c
  - [^abc] ===>Except a and b and c
  - [a-z]==>Any Lower case alphabet symbol
  - [A-Z]===>Any upper case alphabet symbol
  - [a-zA-Z]==>Any alphabet symbol
  - [0-9]==> Any digit from 0 to 9
  - [a-zA-Z0-9]==>Any alphanumeric character
  - [^a-zA-Z0-9]==>Except alphanumeric characters(Special Characters)

Ex:

```
import re
match=re.finditer("[abc]","a7Sb@k9#Az")
#match=re.finditer("[^abc]","a7Sb@k9#Az")
#match=re.finditer("[a-z]","a7Sb@k9#Az")
#match=re.finditer("[A-Z]","a7Sb@k9#Az")
#match=re.finditer("[0-9]","a7Sb@k9#Az")
#match=re.finditer("[a-zA-Z]","a7Sb@k9#Az")
#match=re.finditer("[a-zA-Z0-9]","a7Sb@k9#Az")
#match=re.finditer("[^a-zA-Z0-
9]","a7Sb@k9#Az")
for m in match:
   print(m.start(),"......",m.group())
```

PRE-DEFINED CHARACTER CLASSES:

- \s Space character
- \S Any character except space character
- \d Any digit from 0 to 9
- \D Any character except digit
- \w Any word character [a-zA-Z0-9]
- \W Any character except word character (Special Characters)
- . Any character including special characters

Ex:

```
import re
match=re.finditer("\s","a7Sb @k 9#Az")
#match=re.finditer("\S","a7Sb @k 9#Az")
#match=re.finditer("\d","a7Sb @k 9#Az")
#match=re.finditer("\D","a7Sb @k 9#Az")
#match=re.finditer("\w","a7Sb @k 9#Az")
#match=re.finditer("\W","a7Sb @k 9#Az")
#match=re.finditer(".","a7Sb @k 9#Az")
for m in match:
    print(m.start(),"......",m.group())
```

QUANTIFIERS:

- We can use quantifiers to specify the number of occurrences to match.
  - a --- Exactly one 'a'
  - a+ --- At least one 'a'
  - a* --- Any number of a's including zero number
  - a? --- At most one 'a' i.e. either zero number or one number
  - a{m} --- Exactly m number of a's
  - a{m, n} --- Minimum m number of a's and Maximum n number of a's

Ex:

```
import re
match=re.finditer("a","abaabaaab")
#match=re.finditer("a+","abaabaaab")
#match=re.finditer("a*","abaabaaab")
#match=re.finditer("a?","abaabaaab")
#match=re.finditer("a{2}","abaabaaab")
#match=re.finditer("a{2,3}","abaabaaab")
for m in match:
   print(m.start(),"......",m.group())
```

match ():

- We can use match function to check the given pattern at beginning of target string.
- If the match is available then we will get Match object, otherwise we will get None.

Ex:

```
import re
p=input("Enter pattern to check:")
m=re.match(p,"hyderabad")
if m!= None:
    print("Match is available at the beginning
of the String")
    print("Start Index:",m.start(), "and End
Index:",m.end())
else:
    print("Match is not available at the
beginning of the String")
print(m)
```

fullmatch ():

- We can use fullmatch () function to match a pattern to all of target string. That means complete string should be matched according to given pattern.
- If complete string matched then this function returns match object otherwise it returns None.

Ex:

```
import re
p=input("Enter pattern to check: ")
m=re.fullmatch(p,"hyderabad")
if m!= None:
    print("Full String Matched")
else:
    print("Full String not Matched")
print(m)
```

search ():

- We can use search () function to search the given pattern in the target string.
- If the match is available then it returns the match object which represents first occurrence of the match. If the match is not available then it returns None

Ex:

```
import re
p=input("Enter pattern to check: ")
m=re.search(p,"hyderabad")
if m!= None:
    print("Match is available")
    print("First Occurrence of match with
start index:",m.start(),"and end
index:",m.end())
else:
    print("Match is not available")
print(m)
```

findall ():

- To find all occurrences of the match.
- This function returns a list object which contains all occurrences.

Ex:

```
import re
l=re.findall("[0-9]","a7b9c5kz")
print(l)
```

sub ():

- sub means substitution or replacement
- re.sub(regex,replacement,targetstring)
- In the target string every matched pattern will be replaced with provided replacement.

Ex:

```
import re
s=re.sub("[a-z]","#","a7b9c5k8z")
print(s)
```

subn ():

- It is exactly same as sub except it can also returns the number of replacements.
- This function returns a tuple where first element is result string and second element is number of replacements.

Ex:

```
import re
t=re.subn("[a-z]","#","a7b9c5k8z")
print(t)
print("The Result String:",t[0])
print("The number of replacements:",t[1])
```

split ():

- If we want to split the given target string according to a particular pattern then we should go for split() function.
- This function returns list of all tokens.

Ex:

```
import re
l = re.split(",",
"shashi,nandan,shanvi,mohan,sruthi")
print(l)
for t in l:
    print(t)
```

Ex:

```
import re
l = re.split("\.", "www.durgasoft.com")
print(l)
for t in l:
    print(t)
```

^ Symbol:

- We can use ^ symbol to check whether the given target string starts with our provided pattern or not.

Ex:

```
import re
s="Learning Python is Very Easy"
m=re.search("^Learn",s)
if m != None:
    print("Target String starts with Learn")
else:
    print("Target String Not starts with
Learn")
```

$ Symbol:

- We can use $ symbol to check whether the given target string ends with our provided pattern or not.

Ex:

```
import re
s="Learning Python is Very Easy"
m=re.search("Easy$",s)
if m != None:
    print("Target String Ends with Easy")
else:
    print("Target String Not Ends with Easy")
```

- To ignore the cases then we use third parameter i.e. re.IGNORECASE

Ex:

```
import re
s="Learning Python is Very Easy"
m=re.search("EASY$",s,re.IGNORECASE)
if m != None:
    print("Target String Ends with Easy")
else:
    print("Target String Not Ends with Easy")
```

Write a Regular Expression to represent all 10-digit mobile numbers

Rules:

1. Every number should contains exactly 10 digits
2. The first digit should be 7 or 8 or 9

[7-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]

Or

[7-9][0-9]{9}

Or

[7-9]\d{9}

Or

\d{10}--- to represent 10-digit mobile numbers

Ex:

```
import re
n=input("Enter mobile number:")
m=re.fullmatch("[7-9]\d{9}",n)
if m!= None:
   print("Valid Mobile Number")
else:
   print("Invalid Mobile Number")
```

Write a Python Program to check whether the given mail id is valid gmail id or not

Ex:

```
import re
mailid=input("Enter Mail id:")
m=re.fullmatch("\w[a-zA-Z0-
9_.]*@gmail[.]com",mailid)
if m!=None:
   print("Valid Mail Id");
else:
   print("Invalid Mail id")
```

Note: the above example will work for only gmail id

Validation Expression for all mail ID's

\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*

Ex:

```
import re
mailid=input("Enter Mail id:")
m=re.fullmatch("\w+([-+.']\w+)*@\w+([-
.]\w+)*\.\w+([-.]\w+)*",mailid)
if m!=None:
   print("Valid Mail Id");
else:
   print("Invalid Mail id")
```

PDBC (PYTHON DATABASE CONNECTIVITY):

- As the Part of our Applications, we required to store our Data like Customers Information, Billing Information, Calls Information etc.
- To store this Data, we required Storage Areas.
  There are 2 types of Storage Areas.
  1. Temporary Storage Areas
  2. Permanent Storage Areas

Temporary Storage Areas:

- These are the Memory Areas where Data will be stored temporarily. Ex: Python objects like List, Tuple, and Dictionary.
- Once Python program completes its execution then these objects will be destroyed automatically and data will be lost.

Permanent Storage Areas:

- Here we can store Data permanently.
  Ex: File Systems, Databases, Data warehouses.

Databases:

- We can store Huge Amount of Information in the Databases.
- Query Language Support is available for every Database and hence we can perform Database Operations very easily.
- To access Data present in the Database, compulsory username and password must be required. Hence Data is secured.
- Inside Database Data will be stored in the form of Tables.
- While developing Database Table Schemas, Database Admin follows various Normalization Techniques and can implement various Constraints like Unique Key Constrains, Primary Key Constraints etc. which prevent Data Duplication. Hence there is no chance of Data Inconsistency Problems.

Python Database Programming:

- Sometimes as the part of Programming requirement we have to connect to the database and we have to perform several operations like creating tables, inserting data, updating data, deleting data, selecting data etc.
- We can use SQL Language to talk to the database and we can use Python to send those SQL commands to the database.
- Python provides inbuilt support for several databases like Oracle, MySQL, SqlServer, and SQLite.

Ex to perform CRUD operations using MS-SQL:

```
import pyodbc

def read(con):
    print("Reading data from Database")
    cursor=con.cursor()
    cursor.execute("select*From emp")
    for row in cursor:
        print(row)
    print()

def create(con):
    print("Inserting data into Database")
    cursor=con.cursor()
    cursor.execute("insert into
emp(id,name,address,salary)
values(?,?,?,?);",(3,"mohan","hyd",5000))
    con.commit()
    read(con)

def update(con):
    print("Updating data into Database")
    cursor=con.cursor()
    cursor.execute("update emp set
name=?,address=?,salary=? where
id=?;",("manoj","hyd",23000,1))

    con.commit()
    read(con)
```

```
def delete(con):
    print("Deleting data from Database")
    cursor=con.cursor()
    cursor.execute("delete from emp where id=2")
    con.commit()
    read(con)


con=pyodbc.connect("Driver={SQL
Server};server=.;database=python@8am")

read(con)
create(con)
update(con)
delete(con)
con.close()
```

WORKING WITH MYSQL:

Ex1: program for checking connection

```
import mysql.connector

try:
    con=mysql.connector.connect(
        user='root',
        password='Shanvi@123',
        #host='localhost',
        #port=3306
    )
    if con.is_connected():
        print("Connected successfully")
except:
    print("unable to connect")
```

Ex2: program for creating a database

```
import mysql.connector

try:
    con=mysql.connector.connect(
        user='root',
        password='Shanvi@123',
        host='localhost',
        port=3306
    )
    if con.is_connected():
        print("Connected successfully")
except:
    print("unable to connect")

sql='create database mydb'


cur=con.cursor()
cur.execute(sql)

cur.close()
con.close()
```

Ex3: program for creating a table

```
import mysql.connector

try:
    con=mysql.connector.connect(
        user='root',
        password='Shanvi@123',
        host='localhost',
        database='mydb',
        port=3306
    )
    if con.is_connected():
        print("Connected successfully")


except:
    print("unable to connect")

sql='create table emp(id int,name
varchar(20),address varchar(20))'


cur=con.cursor()
cur.execute(sql)

cur.close()
con.close()
```

Ex4: program for insert a record

```
import mysql.connector

try:
    con=mysql.connector.connect(
        user='root',
        password='Shanvi@123',
        host='localhost',
        database='mydb',
        port=3306
    )
    if con.is_connected():
        print("Connected successfully")
except:
    print("unable to connect")

sql='insert into emp(id,name,address)
values(101,"sai","hyderabad")'


cur=con.cursor()
cur.execute(sql)
con.commit()

cur.close()
con.close()
```

Ex5: program for insert multiple records

```
import mysql.connector

try:
    con=mysql.connector.connect(
        user='root',
        password='Shanvi@123',
        host='localhost',
        database='mydb',
        port=3306
    )
    if con.is_connected():
        print("Connected successfully")
except:
    print("unable to connect")

sql='insert into emp(id,name,address)
values(102,"mohan","sr
nagar"),(103,"shashi","ameerpet")'


cur=con.cursor()
cur.execute(sql)
con.commit()

cur.close()
con.close()
```

Ex6: program for delete record

```
import mysql.connector

try:
    con=mysql.connector.connect(
        user='root',
        password='Shanvi@123',
        host='localhost',
        database='mydb',
        port=3306
    )
    if con.is_connected():
        print("Connected successfully")
except:
    print("unable to connect")

sql='delete from emp where id=101'


cur=con.cursor()
cur.execute(sql)
con.commit()

cur.close()
con.close()
```

Ex7: program for updating a record

```
import mysql.connector

try:
    con=mysql.connector.connect(
        user='root',
        password='Shanvi@123',
        host='localhost',
        database='mydb',
        port=3306
    )
    if con.is_connected():
        print("Connected successfully")
except:
    print("unable to connect")

sql='update emp set
name="manoj",address="secbad" where id=102'


cur=con.cursor()
cur.execute(sql)
con.commit()


cur.close()
con.close()
```

Ex8: program for reading data from database

```
import mysql.connector

try:
    con=mysql.connector.connect(
        user='root',
        password='Shanvi@123',
        host='localhost',
        database='mydb',
        port=3306
    )
    if con.is_connected():
        print("Connected successfully")
except:
    print("unable to connect")

sql='select*from emp'
cur=con.cursor()
cur.execute(sql)

emp_details=cur.fetchall()
#print(emp_details)

for e in emp_details:
    #print(e)
    print("Id:",e[0])
    print("Name:",e[1])
    print("Address:",e[2])

cur.close()
con.close()
```

Ex9: program to perform CRUD operations

```
import mysql.connector

cursor=''
con=''
try:

con=mysql.connector.connect(host='localhost',data
base='python_11am',user='root',password='Shanvi@1
23')
    cursor=con.cursor()
    cursor.execute("create table student(sid
int(5) primary key,sname varchar(10),saddress
varchar(10))")
    print("Table is created")
    sql="insert into student(sid,sname,saddress)
values(%s,%s,%s)"

records=[(101,"sai","hyd"),(102,"manoj","hyd")]
    cursor.executemany(sql,records)
    con.commit()
    print("records inserted successfully")
    cursor.execute("select*from student")
    data=cursor.fetchall()
    for row in data:
        print("Student id:",row[0])
        print("Student name:", row[1])
        print("Student address:", row[2])
        print()
except Exception as e:
    if con==True:
        con.rollback()
        print("There is a problem with sql:",e)
    else:
        print("connection failed",e)
finally:
    if cursor:
        print("finally block")
        cursor.close()
    if con:
        con.close()
```

DECORATOR FUNCTIONS:

- A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure.

- Decorators are usually called before the definition of a function you want to decorate.

- Decorator is a function which can take a function as argument And extend its functionality and returns modified function with Extended functionality.

- Decorators can be extremely useful as they allow the extension of an existing function, without any modification to the original function source code.

- The main objective of decorator functions is we can extend the functionality of existing functions without modifies that function.

Ex:

```
def wish(name):
    print("hello",name,"good morning")

wish("mohan")
wish("durga")
```

- The above function can display always same output for any name
- But we want to modify this function to provide different message if name is "raj"
- We can do this without touching wish () function by using decorator.

Ex:

```
def decor(wish):
    def inner(name):
        if name=="raj":
            print("hello raj good evening")
        else:
            wish(name)
    return inner

@decor
def wish(name):
    print("hello",name,"good morning")

wish("mohan")
wish("durga")
wish("raj")
```

Ex:

```
def mydiv(div):
    def inner(a,b):
        if a<b:
            a,b=b,a
        return  div(a,b)
    return  inner

@mydiv
def div(a,b):
    print(a/b)

div(2,4)
```

Note: we can call same function with decorator and without decorator.

Ex:

```
def decor(wish):
    def inner(name):
        if name=="raj":
            print("hello raj good evening")
        else:
            wish(name)
    return inner

def wish(name):
    print("hello",name,"good morning")

d = decor(wish)

wish("mohan") #decorator will not execute
wish("durga")#decorator will not execute
d("raj") #decorator will execute
```

GENERATOR FUNCTIONS:

- Python provides a generator to create your own iterator function.
- A generator is a special type of function which does not return a single value; instead, it returns an iterator object with a sequence of values.
- In a generator function, a **yield** statement is used rather than a return statement.
- A generator function is defined like a normal function, but whenever it needs to generate a value, it does with yield keyword rather than return.
- If body of **def** contain yield, the function automatically becomes a generator function.

Generator VS Normal collection:

Normal collection:

Ex:

```
l = [x for x in
range(10000000000000000000000)]
for i in l:
    print(i)
```

Note: we will get memory error in this case, because all these values are required to store in the memory.

Generators:

Ex:

```
g = (x for x in
range(10000000000000000000000))
for i in g:
    print(i)
```

Note: we will not get memory error in this case, because all these values are not stored at the beginning.

Note: generators are best suitable for reading data from large files.

- Generator is a function which is used to generate sequence of values.

Ex:

```
def f1():
    yield 123
    yield "sai"
    yield "hyd"
    yield 30000

g = f1()
print(type(g))

print(next(g))
print(next(g))
print(next(g))
print(next(g))
```

Ex:

```
def f1():
    yield 123
    yield "sai"
    yield "hyd"
    yield 30000

for i in f1():
    print(i)
```

YIELD KEYWORD:

- Yield is a keyword in Python that is used to return from a function without destroying the states of its local variable and when the function is called, the execution starts from the last yield statement. Any function that contains a yield keyword is termed a generator.

ASSERTIONS:

- The assert keyword is used when debugging code.
- The assert keyword lets you test if a condition in your code returns True, if not, the program will raise an Assertion Error. You can write a message to be written if the code returns False.

Debugging python program by using assert keyword:

- The process of identifying and fixing the bug is called debugging.
- The very common way of debugging is to use print function.
- But the problem with the print function is after fixing the bug, compulsory we have to delete extra added print statements ,otherwise these statements will be execute at runtime performance problems and disturb the console output window.
- To overcome this problem we should go for assert statement.
- The main advantage of assert statement over the print is after fixing the bug we are not required to delete assert statements.
- The main purpose of assertions is to perform debugging.
- Assertions concept is applicable for development and testing environment but not for production environment.

  Syntax: assert conditionl_expression, message

- Conditionl_expression will be evaluated and if it is true then the program will continue. If it is false then the program will be terminated by raising assertion error.
- By seeing assertion error, programmer can analyse the code and fix the problem.

Ex:

```
def square(x):
    return x*x

assert square(3)==9,"The square of 3 should be
9"
assert square(4)==16,"The square of 4 should
be 16"
assert square(5)==25,"The square of 3 should
be 25"

print(square(3))
print(square(4))
print(square(5))
```

Note: in the above program we will get assertion error.

Ex:

```
def square(x):
    return x*x

assert square(3)==9,"The square of 3 should be
9"
assert square(4)==16,"The square of 4 should
be 16"
assert square(5)==25,"The square of 3 should
be 25"

print(square(3))
print(square(4))
print(square(5))
```

Note: Assertions concept can be used to resolve the development errors but exception handling can be used to handle runtime errors.

LOGGING IN PYTHON:

- Logging is a means of tracking events that happen when some software runs.
- Logging is important for software developing, debugging, and running. If you don't have any logging record and your program crashes, there are very few chances that you detect the cause of the problem.
- Logging is a very useful tool in a programmer's toolbox. It can help you develop a better understanding of the flow of a program and discover scenarios that you might not even have thought of while developing.
- Logs provide developers with an extra set of eyes that are constantly looking at the flow that an application is going through. They can store information.
- It is highly recommended to store complete application flow and exceptions information to a file. This process is called logging.
- The main advantages of logging are we can use log files while performing debugging.

LOGGING LEVELS:

- Depending on type of information, logging data is divided according to the following 6 levels in Python.
- **CRITICAL**==>50==> represents a very serious problem that needs high attention
- **ERROR**===>40===> represents a serious error
- **WARNING**==>30==> represents a warning message, some caution needed. It is alert to the programmer

- **INFO**===>20===> represents a message with some important information.
- **DEBUG**===>10==>represents a message with debugging information
- **NOTSET**==>0==> represents that the level is not set.
- By default while executing python program only WARNING and higher level messages will be displayed.

How to implement logging:

- To perform logging, first we required create a file to store messages and we have to specify which level messages we have to store.
- We can do this by using **basicconfig ()** function of logging module.

Ex:

```
import logging

logging.basicConfig(filename='log.txt',
level=logging.WARNING)
print("Logging Module Demo")
logging.debug("This is debug message")
logging.info("This is info message")
logging.warning("This is warning message")
logging.error("This is error message")
logging.critical("This is critical message")
```

Note: In the above program only WARNING and higher level messages will be written to log file.

If we set level as DEBUG then all messages will be written to log file.

Ex:

```
import logging

logging.basicConfig(filename='log.txt',
level=logging.DEBUG)
print("Logging Module Demo")
logging.debug("This is debug message")
logging.info("This is info message")
logging.warning("This is warning message")
logging.error("This is error message")
logging.critical("This is critical message")
```

Program to write exception information to the log file

```
import logging

logging.basicConfig(filename='mylog.txt',level
=logging.INFO)

logging.info("A New request Came:")
try:
        x=int(input("Enter First Number: "))
        y=int(input("Enter Second Number: "))
        print(x/y)

except ZeroDivisionError as msg:
        print("cannot divide with zero")
        logging.exception(msg)

except ValueError as msg:
        print("Enter only int values")
        logging.exception(msg)

logging.info("Request Processing Completed")
```

PACKAGES IN PYTHON:

- A package is a folder or directory of Python modules.
- We usually organize our files in different folders and subfolders based on some criteria, so that they can be managed easily and efficiently.
- For example, we keep all our games in a Games folder and we can even subcategorize according to the genre of the game or something like this. The same analogy is followed by the Python package.
- A Python module may contain several classes, functions, variables, etc. whereas a Python package can contains several modules.
- In simpler terms a package is folder that contains various modules as files.

ACCESSING MODULES FROM PACKAGE:

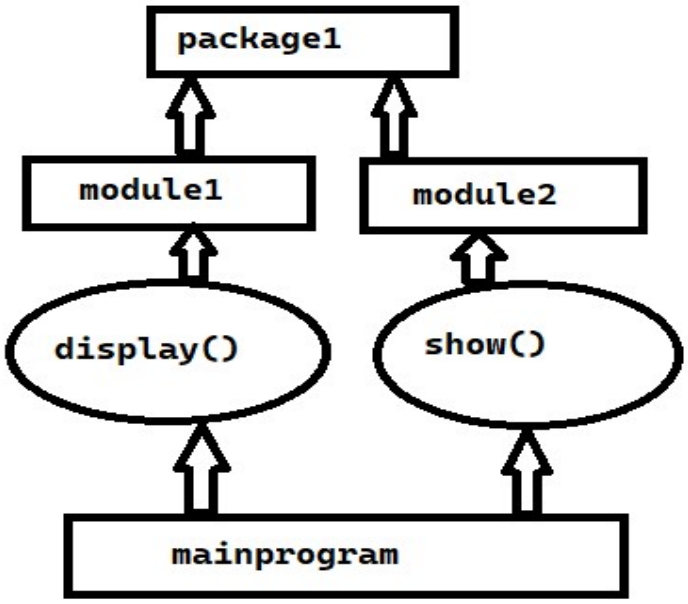
- To access or consume modules from packages then we use the following syntax.

  Syntax:

  Import sys

  sys.path.append("path of the package")

Ex: Importing modules from single package



- **Steps to create a package in pycharm editor**
    - Right Click on your project
    - Go to New
    - Click on python package
    - Give the name package1
- **Steps to create a modules in package1**
    - Right click on package1 folder
    - Go to New
    - Click on python file
    - Give the name module1
    - Create another module with the name module2

module1.py:

```
def display():
    print("display function from module1")
```

module2.py:

```
def show():
    print("show function from module2")
```

- **Steps to create a main program**

    - Right Click on your project
    - Go to New
    - Click on python file
    - Give the name mainprogram

mainprogram.py:

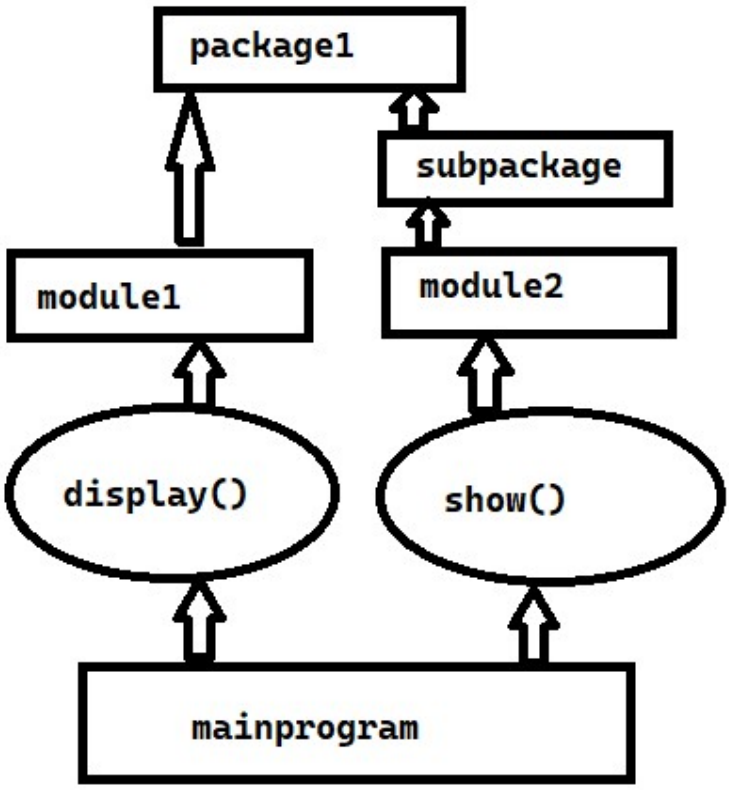
```
import sys
sys.path.append("F:/pythonProject@8am/package1
")

#option1
import module1
import module2

module1.display()
module2.show()

#option2
from module1 import *
from module2 import *

display()
show()
```

- **Steps to add path of the package into main program**

    - Right click on package1 folder
    - Click on Copy path/Reference
    - Click on Absolute path
    - Then paste into append method

- Now run the manprogram and check.

Ex: Importing modules from sub package



module1.py:

```
def display():
    print("display function from module1-
package1")
```

module2.py:

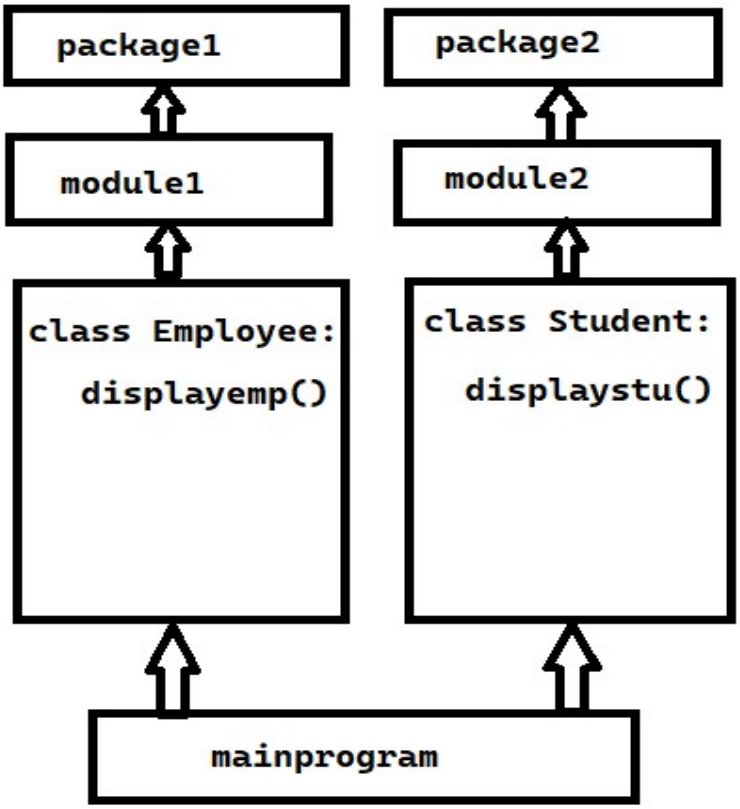
```
def show():
    print("show function from module2-
subpackage-package1")
```

mainprogram.py:

```
import sys
sys.path.append("F:/pythonProject@8am/package1")

from module1 import *

display()

sys.path.append("F:/pythonProject@8am/package1/
subpackage")
from module2 import *
show()
```

- Now run the manprogram and check.

Ex: Importing classes from 2 different modules and packages



module1.py:

```
class Employee:
    def __init__(self,eid,ename,eaddress):
        self.eid=eid
        self.ename=ename
        self.eaddress=eaddress

    def displayemp(self):

print(self.eid,self.ename,self.eaddress)
```

module2.py:

```
class Student:
    def __init__(self,sid,sname,saddress):
        self.sid=sid
        self.sname=sname
        self.saddress=saddress

    def displaystu(self):

print(self.sid,self.sname,self.saddress)
```

mainprogram.py:

```
import sys
sys.path.append("F:\pythonProject@8am\package1")

from module1 import Employee
e=Employee(101,"sai","hyderabad")
e.displayemp()

sys.path.append("F:\pythonProject@8am\package2")

from module2 import Student
s=Student(1234,"raj","sr nagar")
s.displaystu()
```

MULTI-THREADING IN PYTHON:

Multi-tasking:

- Executing several tasks simultaneously is called multitasking.
- There are 2 types of multitasking

    1. Process based multitasking---multi processing
    2. Thread based multitasking----multi-threading

Process based multitasking:

- Executing several tasks simultaneously where each task is a separate independent process is called process based multitasking.

- **Ex:** while typing python program in pycharm editor, we can listen mp3 songs from same system, at the same we can also download a file from internet.

- All these tasks are executing simultaneously and independent of each other this type of multi-tasking is best suitable at operating system level.

Thread based multitasking:

- Executing several tasks simultaneously where each task is a separate Independent part of the same system is called thread based multitasking and each independent part is called a thread.

- This type of multi-tasking is best suitable at programmatic level.

- **Note:** whether it is process based or thread based, the main advantage of multi-tasking is to improve performance of the system by reducing response time

- Application area where we use multi-threading concept:
  - To implement multimedia graphics
  - To develop animations
  - To develop video games
  - To develop web application servers
- Python provides one inbuilt module called **threading** to provide support for developing threads.
- Every python program by default contains one thread which is nothing but Main Thread.

Ex:

```
import threading

print("Current Executing
Thread:",threading.current_thread().getName())
threading.current_thread().setName("MohanThrea
d")
print("Current Executing
Thread:",threading.current_thread().getName())
```

Note: In the above program we will get warnings, to avoid warnings then we use the following code.

Ex:

```
import warnings
warnings.filterwarnings("ignore",
category=DeprecationWarning)
from threading import *

print("Current Executing
Thread:",current_thread().getName())
current_thread().setName("MohanThread")
print("Current Executing
Thread:",current_thread().getName())
```

- **Deprecation warnings** are a common thing in IT industry.
- They are warnings that notify us that a specific feature (e.g. a method) will be removed soon (usually in the next minor or major version) and should be replaced with something else.

Difference between single thread and multi-thread program

Program with single thread (MainThread):

```
import time

def square(numbers):
    for n in numbers:
        time.sleep(1)
        print("square is:",n*n)

def cube(numbers):
    for n in numbers:
        time.sleep(1)
        print("cube is:",n*n*n)


numbers=[2,3,4,5,6]
t=time.time()
square(numbers)
cube(numbers)
print("Done in:",time.time()-t)
#Done in: 10.334125518798828
```

Program with Multi-Thread:

```
import time
import threading

def square(numbers):
    for n in numbers:
        time.sleep(1)
        print("square is:",n*n)
def cube(numbers):
    for n in numbers:
        time.sleep(1)
        print("cube is:",n*n*n)

numbers=[2,3,4,5,6]
t=time.time()

t1=threading.Thread(target=square,args=(number
s,))
t2=threading.Thread(target=cube,args=(numbers,
))

t1.start()
t2.start()

t1.join()
t2.join()

print("Done in:",time.time()-t)
#Done in: 5.050450086593628
```

Note: multi-thread programs will reduce the execution time.

THE WAYS OF CREATING THREADS IN PYTHON:

- Creating a thread without using any class
- Creating a thread by extending a thread class
- Creating a thread without extending thread class

Creating a thread without using any class

Ex :

```
from threading import *

def display():
    for i in range(10):
        print("Child Thread")

t=Thread(target=display)
t.start()


#below code executed by main Thread
for i in range(10):
    print("Main Thread")
```

Note: if multiple threads present in our program, then we cannot expect execution order and we cannot expect exact output for the multi-threaded programs

- Thread is a predefined class which is present in threading module it can be used to create our own threads.

CREATING A THREAD BY EXTENDING A THREAD CLASS

- We have to create child class for Thread class, in that child class we have to override run() method with our required job
- Whenever we call start () method then automatically run () will be executed and performs our job.

Ex :

```
from threading import *

class Test(Thread):
    def run(self):
        for i in range(10):
            print("Child Thread")

t=Test()
t.start()
```

- **Creating a thread without extending a thread class**

Ex :

```
from threading import *

class Test:
    def display(self):
        for i in range(10):
            print("Child Thread")

t=Test()

t1=Thread(target=t.display)
t1.start()
```

Thread identification number:

- For every thread internally a unique identification number is available we can access this id by using implicit variable i.e. "**ident**"

Ex :

```
from threading import *

def f1():
    print("Child Thread")

t=Thread(target=f1)
t.start()
print("Main Thread Identification
number:",current_thread().ident)
print("Child Thread Identification
number:",t.ident)
```

active_count ():

- This function returns the no of active threads currently running.

Ex :

```
from threading import *
import time

def f1():

print(current_thread().getName(),"...started")
time.sleep(3)

print(current_thread().getName(),"...ended")
print("The no of active threads:",active_count())

t1=Thread(target=f1,name="childthread1")
t2=Thread(target=f1,name="childthread2")
t3=Thread(target=f1,name="childthread3")
t1.start()
t2.start()
t3.start()

print("The no of active threads:",active_count())
time.sleep(5)
print("The no of active threads:",active_count())
```

is_alive ():

- This method is used to check whether a thread is still executing or not

Ex :

```
from threading import *
import time

def f1():

print(current_thread().getName(),"..started")
    time.sleep(3)

print(current_thread().getName(),"...ended")

t1=Thread(target=f1,name="childthread1")
t2=Thread(target=f1,name="childthread2")
t1.start()
t2.start()

print(t1.name,"is Alive:",t1.is_alive())
print(t2.name,"is Alive:",t2.is_alive())
time.sleep(5)
print(t1.name,"is Alive:",t1.is_alive())
print(t2.name,"is Alive:",t2.is_alive())
```

Join ():

- If thread wants to wait until completing some other thread then we should go for join () method.

Ex :

```
from threading import *
import time

def f1():
    for i in range(10):
        print("childthread")
        time.sleep(1)

t=Thread(target=f1)
t.start()

#tbelow code executed by main thread
#t.join()
t.join(5)
for i in range(10):
    print("mainthread")

#note: we can call join() with time period
also
```

THREAD SYNCHRONIZATION:

- If multiple threads are executing simultaneously then there may be a chance of data inconsistency problems.

Ex :

```
from threading import *
import time

def f1(name):
    for i in range(10):
        print("Good morning:",end="")
        time.sleep(2)
        print(name)

t1=Thread(target=f1,args=("mohan",))
t2=Thread(target=f1,args=("sai",))
t1.start()
t2.start()
```

- We are getting irregular output because both threads are executing
- simultaneously for f1() , to overcome this problem we should go for synchronization

- In Synchronization the threads will execute one by one so that we can overcome data inconsistency problems.
- Synchronization means at a time only one thread

- In python we can implement Synchronization by using the following technique.
    1. Lock
    2. RLock
    3. Semaphore

Synchronization by using Lock:

- Locks are the most fundamental Synchronization mechanism provided by threading module.
- We can create Lock object like: l=Lock()
- The lock object can be hold by only one thread at a time, if any other thread required the same lock then it will wait until thread releases lock.
- A thread can acquire the lock by using acquire() method
  Ex: l.acquire()
- A thread can release the lock by using release() method
  Ex: l.release ()

Note: To call release () method compulsory thread should be owner of that lock that means thread should has the lock already, otherwise we will get runtime error.

Ex:

```
from threading import *
l=Lock()
#l.acquire()  #RuntimeError: release unlocked lock
l.release()
```

Ex :

```
from threading import *
import time

l=Lock()
def f1(name):
    l.acquire()
    for i in range(10):
        print("Good morning:",end="")
        time.sleep(2)
        print(name)
    l.release()

t1=Thread(target=f1,args=("mohan",))
t2=Thread(target=f1,args=("sai",))
t1.start()
t2.start()
```

The problem with simple Lock:

- The standard lock object doesn't know which thread is currently holding that lock.
- If the lock is held and any thread attempts to acquire the lock, then it will be blocked even the same thread is already holding that lock.
- To overcome this then we use RLock.

Synchronization by using RLock:

- RLock means Reentrant lock.
- Reentrant lock means the thread can acquire the lock again and again
- That means same thread can acquire the lock any no of times.
- In case any other thread acquires the lock then it will be blocked.

Ex:

```
from threading import *
l=RLock()
def factorial(n):
    l.acquire()
    if n==0:
        result=1
    else:
        result=n*factorial(n-1)
    l.release()
    return result

def res(n):
    print("The factorial
of",n,"is:",factorial(n))

t1=Thread(target=res,args=(4,))
t2=Thread(target=res,args=(5,))
t1.start()
t2.start()
```

Synchronization by using semaphore:

- In case of Lock and RLock at a time only one thread is allowed to execute
- Sometimes our requirement is at a time particular no of threads are allowed to execute in this case then we use semaphore.
- Semaphore is advanced synchronization mechanism.
- We create semaphore objects like:
  Ex : s=Semaphore(counter)
- Here counterpart represents maximum number of threads is allowed to executes simultaneously.
- The default value of counterpart is 1
- Whenever thread executes acquire() method then the counterpart value will be decremented by 1 and if thread execute release() method then counterpart value will be increment by 1

Ex:

```
from threading import *
import time

s=Semaphore(2)

def f1(name):
    s.acquire()
    for i in range(10):
        print("Good morning:",end="")
        time.sleep(2)
        print(name)
    s.release()

t1=Thread(target=f1,args=("sai",))
t2=Thread(target=f1,args=("mohan",))
t3=Thread(target=f1,args=("kiran",))
t4=Thread(target=f1,args=("raj",))

t1.start()
t2.start()
t3.start()
t4.start()
```

Bounded semaphore:

- Normal semaphore is an unlimited semaphore which allows us to call release () method any no of times to increment counter.
- The no of release () method calls can exceed the no of acquire () method calls.

Ex:

```
from threading import *
s=Semaphore(2)
s.acquire()
s.acquire()
s.release()
s.release()
s.release()
print("End of the program")
```

- Bounded semaphore exactly same as normal semaphore but in bounded semaphore the no of release () method calls should not exceed the no of acquire () method calls, otherwise we will get value error i.e. semaphore released too many times.

Ex:

```
from threading import *
s=BoundedSemaphore(2)
s.acquire()
s.acquire()
s.release()
s.release()
s.release()
print("End of the program")
#ValueError: Semaphore released too many times
```

WORKING WITH PANDAS:

- Pandas are a software library written for the Python programming language for data manipulation and analysis.
- In particular, it offers data structures and operations for manipulating numerical tables and time series.
- Pandas are defined as an open-source library that provides high-performance data manipulation in Python. The name of Pandas is derived from the word Panel Data.
- **Data Representation:** It represents the data in a form that is suited for data analysis through its Data Frame and Series.

Series:

- It is defined as a one-dimensional array that is capable of storing various data types. The row labels of series are called the index.
- We can easily convert the list, tuple, and dictionary into series using "series' method. A Series cannot contain multiple columns. It has one parameter.
- **Data**: It can be any list, dictionary, or scalar value.

Data Frame:

- It is a widely used data structure of pandas and works with a two-dimensional array with labelled axes (rows and columns).
- Data Frame is defined as a standard way to store data and has two different indexes, i.e., row index and column index.

Ex:

```
import pandas
s=pandas.Series()
print(s)
```

Ex:

```
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data,index=[100,101,102,103])
print(s)
```

Ex:

```
import pandas as pd
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data,index=['b','c','d','a'])
print(s)
```

Ex:

```
import pandas as pd
s = pd.Series(5, index=[0, 1, 2, 3])
print(s)
```

Ex:

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index =
['a','b','c','d','e'])
print(s)
print(s['a'])
print (s[:3])
```

Ex:

```
import pandas as pd
df = pd.DataFrame()
print(df)
```

Ex:

```
import pandas as pd
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print(df)
```

Ex:

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df =
pd.DataFrame(data,columns=['Name','Age'],dtype
=float)
print(df)
```

Ex:

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve',
'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data)
print(df)
#df.tail(2)
df.head(1)
```

Ex:

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve',
'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data,
index=['rank1','rank2','rank3','rank4'])
print(df)
```

Ex:

```
import pandas as pd
df1 = pd.DataFrame([[1, 2], [3, 4]], columns =
['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns =
['a','b'])
df1 = df1.append(df2)
print(df1)
```

Ex:

```
import pandas as pd
df1 = pd.DataFrame({'name': ['John',
'Smith','Paul'],
                    'Age': ['25', '30',
'50']},
                   index=[0, 1, 2])
df2 = pd.DataFrame({'name': ['Adam', 'Smith'
],
                    'Age': ['26', '11']},
                   index=[3, 4])

df_concat = pd.concat([df1,df2])
print(df_concat)
```

Ex:

```
#convert from csv into html
import pandas as pd
data=pd.read_csv("G:/python@9am/emp.csv")
data.to_html('durga.html')
```

WORKING WITH MATPLOTLIB:

- Matplotlib is a cross-platform, data visualization and graphical plotting library for Python and its numerical extension NumPy.
- Matplotlib is a Python package used for data plotting and visualisation.
- It is a useful complement to Pandas, and like Pandas, is a very feature-rich library which can produce a large variety of plots, charts, maps, and other visualisations

Ex:

```
from matplotlib import pyplot as plt

# x-axis values
x = [5, 2, 1, 4, 7]
# Y-axis values
y = [10, 5, 8, 4, 2]
# Function to plot
plt.plot(x,y,marker='o')
# function to show the plot
plt.show()
```

Ex:

```
from matplotlib import pyplot as plt
x=[5,8,10]
y=[12,16,6]
plt.plot(x,y)
plt.title("Simple Graph")
plt.ylabel("Y-axis")
plt.xlabel("X-axis")
plt.show()
```

Ex:

```
from matplotlib import pyplot as plt

x=[5,8,10]
y=[12,16,6]
x2=[6,9,11]
y2=[6,15,7]
plt.plot(x,y,'g',label="line one",linewidth=3)
plt.plot(x2,y2,'r',label="line
two",linewidth=5)
plt.title("Simple Graph")
plt.ylabel("Y axis")
plt.xlabel("X axis")
plt.legend()
plt.grid(True,color='b')
plt.show()
```

Ex:

```
from matplotlib import pyplot as plt

plt.bar([1,3,5,7,9],[5,2,7,8,2],label="Bar
one",color='y')
plt.bar([2,4,6,8,10],[8,6,2,5,6],label="Bar
two",color='g')
plt.legend()
plt.xlabel('bar number')
plt.ylabel('bar height')
plt.title("Bar Graph")
plt.show()
```

Ex:

```
from matplotlib import pyplot as plt
x=[1,2,3,4,5,6,7,8]
y=[5,2,4,2,1,4,5,2]
plt.scatter(x,y,color='r',s=80,marker="*")
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title("Scatter Plot")
plt.show()
```

Ex:

```
from matplotlib import pyplot as plt

days=[1,2,3,4,5]
sleeping=[7,8,6,11,7]
eating=[2,3,4,3,2]
working=[7,8,7,2,2]
playing=[8,5,7,8,13]

plt.plot([],[],color='m',label='sleeping',line
width=5)
plt.plot([],[],color='c',label='eating',linewi
dth=5)
plt.plot([],[],color='r',label='working',linew
idth=5)
plt.plot([],[],color='g',label='playing',linew
idth=5)

plt.stackplot(days,sleeping,eating,working,pla
ying,colors=['m','c','r','k'])
plt.xlabel('X')
plt.ylabel('Y')
plt.title("Stack plot")
plt.legend()
plt.show()
```

Ex:

```
from matplotlib import pyplot as plt

parts = [8, 2, 10, 4]
activities = ['sleeping', 'eating', 'working',
'playing']
cols = ['c', 'm', 'r', 'b']
plt.pie(parts,
        labels=activities,
        colors=cols,
        startangle=90,
        shadow=True,
        explode=(0, 0.1, 0, 0),
        autopct='%1.1f%%'
        )
plt.title("PiePlot")
plt.show()
```

ALL THE BEST