
python

UNIT – IV

- **Advance Python- OOPs concept**

  Class and object

  Attributes

  Inheritance

  Overloading and Overriding,

  Data hiding.

- **Regular expressions**

  Match function

  Search function,

  Matching VS Searching,

  Modifiers Patterns.

Python OOPs Concepts

- Like other general purpose languages, python is also an object-oriented language since its beginning. Python is an object-oriented programming language. It allows us to develop applications using an Object Oriented approach. In Python, we can easily create and use classes and objects.
- Major principles of object-oriented programming system are given below.

- Object
- Class
- Method
- Inheritance
- Polymorphism
- Data Abstraction
- Encapsulation

Object-oriented vs Procedure-oriented Programming languages

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

Class:

- In python every thing is an object.
- To create objects we required some Model or Plan or Blue print, which is nothing but class.
- We can write a class to represent properties (attributes) and actions (behaviour) of object.
- Properties can be represented by variables
- Actions can be represented by Methods.
- Hence class contains both variables and methods.

How to Define a class?

We can define a class by using class keyword.

Syntax:

```
class className:
    ''' documentation string '''
    variables: instance variables, static and local variables
    methods: instance methods, static methods, class methods
```

- Documentation string represents description of the class. Within the class doc string is always optional.
- We can get doc string by using the following 2 ways.
    1. print(classname.__doc__)
    2. help(classname)
- Within the Python class we can represent data by using variables.
- There are 3 types of variables are allowed.
    1. Instance Variables (Object Level Variables)
    2. Static Variables (Class Level Variables)
    3. Local variables (Method Level Variables)
- Within the Python class, we can represent operations by using methods. The following are various types of allowed methods
    1.Instance Methods
    2. Class Methods
    3. Static Method

What is Object:

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

Syntax :

reference variable=class name()

Example:

s = Student()

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

Ex

```
Class Student:
        def __init__(self):
                self.name='John'
                self. age=20
                self.marks=900
        def  talk(self):
                print("Hi, I am",self.name)
                print("my age is",self.age)
                print("my marks are",self.marks)
s=Student()
s.talk()
```

Output

```
Hi, I am John
my age is 20
my marks are 900
```

Self variable:

- self is the default variable which is always pointing to current object(like this keyword in Java)
- By using self we can access instance variables and instance methods of object.
- self should be first parameter inside constructor
- **def __init__(self):**
- self should be first parameter inside instance methods
- **def display(self):**
- Self is parameter in method and we can use another name inplace of self
- **def display(my):**

Constructor:

- Constructor is a special method in python.
- The name of the constructor should be __init__(self)
- Constructor will be executed automatically at the time of object creation.
- The main purpose of constructor is to declare and initialize instance variables.
- Per object constructor will be executed only once.
- Constructor can take at least one argument(atleast self)
- Constructor is optional and if we are not providing any constructor then python will provide default constructor.

Example:

```
def__init__(self,name,rollno,marks):
        self.name=name
        self.rollno=rollno
        self.marks=marks
```

program to demonstrate constructor will execute only once per object:

```
class Test:
    def __init__(self):
        print("Constructor exeuction...")
    def m1(self):
        print("Method Execution...")

t1=Test()
t2=Test()
t3=Test()
t1.m1()
```

```
>>>
Constructor exeuction...
Constructor exeuction...
Constructor exeuction...
Method Execution...
>>>
```

```
class Student:
    '''''Thisisstudentclasswithrequireddata'''
    def __init__(self,x,y,z):
        self.name=x
        self.rollno=y
        self.marks=z
    def display(self):
        print("StudentName:{}\nRollno:{}\nMarks:{}".format(self.name,self.rollno,self.marks))
s1=Student("Ramesh",1216,80)
s1.display()
s2=Student("Suresh",1218,75)
s2.display()
```

```
>>>
StudentName:Ramesh
Rollno:1216
Marks:80
StudentName:Suresh
Rollno:1218
Marks:75
>>>
```

- **Python Constructor**

- A constructor is a special type of method (function) which is used to initialize the instance members of the class.
- Constructors can be of two types.
- Parameterized Constructor
- Non-parameterized Constructor

```
#Python Non-Parameterized Constructor Example

class Student:
    # Constructor - non parameterized
    def __init__(self):
        print("This is non parametrized constructor")
    def show(self,name):
        print("Hello",name)

student = Student()
student.show("Ram")
```

```
This is non parametrized constructor
Hello Ram
>>>
```

```python
#Python Parameterized Constructor Example
class Student:
    # Constructor - parameterized
    def __init__(self, name):
        print("This is parametrized constructor")
        self.name = name
    def show(self):
        print("Hello",self.name)


student = Student("Sam")
student.show()
```

```
This is parametrized constructor
Hello Sam
>>>
```

Differences between Methods and Constructors:

Method	Constructor
1. Name of method can be any name	1. Constructor name should be always __init__
2. Method will be executed if we call that method	2. Constructor will be executed automatically at the time of object creation.
3. Per object, method can be called any number of times.	3. Per object, Constructor will be executed only once
4. Inside method we can write business logic	4. Inside Constructor we have to declare and initialize instance variables

Types of Variables:

Inside Python class 3 types of variables are allowed.

1. Instance Variables(Object Level Variables)
2. Static Variables(Class Level Variables)
3. Local variables (Method Level Variables)

Instance Variables(Object Level 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.

```
class Student:
        def __init__(self,r,n):
            self.rno=r
            self.name=n
    def dis(self):
        print(self.rno,self.name)

s1=Student(12,"IT")
s2=Student(3,"Mech")
s1.dis()
s2.dis()
```

we can declare Instance variables:

1. Inside Constructor by using self variable
2. Inside Instance Method by using self variable
3. Outside of the class by using object reference variable

Example: class Test:

```
    def __init__(self):
            self.a=2
            self.b=3
            def meth(self):
            self.c=4
    t1=Test()
    print("t1's values",t1.__dict__)
    t1.d=5
    print("t1's values",t1.__dict__)
```

_dict__ : Dictionary containing the class's namespace.

How to access Instance variables:

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

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

    def display(self):
            print(self.a)
            print(self.b)
t=Test()
t.display()
print(t.a,t.b)
```

```
10
20
10 20
>>>
```

How to delete instance variable from the object:

1. Within a class we can delete instance variable as follows **del self.variableName**
2. From outside of class we can delete instance variables as follows **del objectreference.variableName**

```
class Test:
    def __init__(self):
        self.a=10
        self.b=20
        self.c=30
        self.d=40
    def m1(self):
        del self.d
t=Test()
print(t.__dict__)
t.m1()
print(t.__dict__)
del t.c
print(t.__dict__)
```

```
{'d': 40, 'b': 20, 'c': 30, 'a': 10}
{'b': 20, 'c': 30, 'a': 10}
{'b': 20, 'a': 10}
>>>
```

Note: The instance variables which are deleted from one object,will not be deleted from other objects.

```
class Test:
    def __init__(self):
        self.w=10
        self.x=20
        self.y=30
        self.z=40

t1=Test()
t2=Test()
del t1.w
```

```
{'y': 30, 'x': 20, 'z': 40}
{'y': 30, 'w': 10, 'x': 20, 'z': 40}
>>>
```

- **If we change the values of instance variables of one object then those changes won't be reflected to the remaining objects, because for every object we are separate copy of instance variables are available.**

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

t1=Test()
t1.a=1216
t1.b=1218
t2=Test()
print('t1:',t1.a,t1.b)
print('t2:',t2.a,t2.b)
```

```
t1: 1216 1218
t2: 10 20
>>>
```

2.Static Variables(Class Level Variables):

- If the value of a variable is not varied from object to object, such type of variables we have to declare with in the class directly but outside of methods. Such type of variables are called Static variables.
- For total class only one copy of static variable will be created and shared by all objects of that class.
- We can access static variables either by class name or by object reference. But recommended to use class name.

Instance Variable vs Static Variable:

- Note: In the case of instance variables for every object a separate copy will be created, but in the case of static variables for total class only one copy will be created and shared by every object of that class.

```
class Test:
    x=10
    def __init__(self):
        self.y=2

t1=Test()
t2=Test()
print('t1:',t1.x,t1.y)
print('t2:',t2.x,t2.y)
Test.x=888
t1.y=999
print('t1:',t1.x,t1.y)
print('t2:',t2.x,t2.y)
```

Various places to declare static variables:

1. In general we can declare within the class directly but from out side of any method
2. Inside constructor by using class name
3. Inside instance method by using class name
4. Inside class method by using either class name or cls variable
5. Inside static method by using class name.

```
class Test:
    a=10
    def __init__(self):
        print(self.a)
        print(Test.a)
    def m1(self):
        print(self.a)
        print(Test.a)
    @classmethod
    def m2(cls):
        print(cls.a)
        print(Test.a)
    @staticmethod
    def m3():
        print(Test.a)
t=Test()
print(Test.a)
print(t.a)
t.m1()
t.m2()
t.m3()
```

```
10
10
10
10
10
10
10
10
>>>
```

Where we can modify the value of static variable:

Anywhere either with in the class or outside of class we can modify by using classname.But inside class method, by using cls variable.

```
class Test:
    a=1312
    @classmethod
    def m1(cls):
        cls.a=1213
    @staticmethod
    def m2():
        Test.a=1216
print(Test.a)
Test.m1()
print(Test.a)
Test.m2()
print(Test.a)
```

```
1312
1213
1216
>>>
```

If we change the value of static variable by using either self or object reference variable:

If we change the value of static variable by using either self or object reference variable, then the value of static variable won't be changed, just a new instance variable with that name will be added to that particular object.

```
class Test:
    a=10
    def m1(self):
        self.a=1312
t1=Test()
t1.m1()
print(Test.a)
print(t1.a)
```

```
10
1312
>>>
```

How to delete static variables of a class:

- We can delete static variables from anywhere by using the following syntax

del classname.variablename

- But inside class method we can also use cls variable

del cls.variablename

- Note: By using object reference variable/self we can read static variables, but we cannot modify or delete.
- If we are trying to modify, then a new instance variable will be added to that particular object.t1.a=16
- If we are trying to delete then we will get error.
- We can modify or delete static variables only by using classname or cls variable.

Local variables:

- Sometimes to meet temporary requirements of programmer, we can declare variables inside a method directly,
- such type of variables are called local variable or temporary variables.
- Local variables will be created at the time of method execution and destroyed once method completes.
- Local variables of a method cannot be accessed from outside of method.

```
class Test:
    def m1(self):
        a=1000
        print(a)
    def m2(self):
        b=2000
        print(b)
t=Test()
t.m1()
t.m2()
```

```
1000
2000
>>>
```

```
class Test:
    def m1(self):
        a=1000
        print(a)
    def m2(self):
        b=2000
        print(b)
        print(a)#NameError:name'a'is notdefined

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

```
1000
2000
Traceback (most recent call last):
  File "C:/Python34/OOPSPROG/16.py", line 12, in <module>
    t.m2()
  File "C:/Python34/OOPSPROG/16.py", line 8, in m2
    print(a)#NameError:name'a'isnotdefined
NameError: name 'a' is not defined
>>>
```

Types of Methods:

Inside Python class 3 types of methods are allowed

1. Instance Methods
2. Class Methods
3. Static Methods

1. Instance Methods:

- Inside method implementation if we are using instance variables then such type of methods are called instance methods.
- Inside instance method declaration, we have to pass self variable.

def m1(self)

- By using self variable inside method we can able to access instance variables.
- Within the class we can call instance method by using self variable and from outside of the class we can call by using object reference.

```
class Student:
    def __init__(self,name,marks):
        self.name=name
        self.marks=marks
    def display(self):
        print('Hi',self.name)
        print('Your Marksare:',self.marks)
    def grade(self):
        if self.marks>=60:
            print('You got FirstGrade')
        elif self.marks>=50:
            print('You                    de')
        elif self.mark
            print('You                    ')
        else:
            print('You
n=int(input('Enter num                     :'))
for i in range(n):
    name=input('Enter
    marks=int(input('Enter Marks:'))
    s=Student(name,marks)
    s.display()
    s.grade()
    print()
```

```
Enter number of students:2
Enter Name:Ramesh
Enter Marks:32
Hi Ramesh
Your Marksare: 32
You are Failed

Enter Name:suresh
Enter Marks:78
Hi suresh
Your Marksare: 78
You got FirstGrade
```

2. Class Methods:

- Inside method implementation if we are using only class variables(static variables),then such type of methods we should declare as class method.
- We can declare class method explicitly by using **@classmethod** decorator.
- For class method we should provide cls variable at the time of declaration
- We can call classmethod by using classname or object reference variable.

```
class Animal:
    legs=4
    @classmethod
    def walk(cls,name):
        print('{} walks with {} legs...'.format(name,cls.legs))
Animal.walk('Dog')
Animal.walk('Cat')
```

```
Dog walks with 4 legs...
Cat walks with 4 legs...
>>>
```

- **Program to track the number of objects created**

```
class Test:
    count=0
    def __init__(self):
        Test.count=Test.count+1
    @classmethod
    def noOfObjects(cls):
        print('The number of objects created for testclass:',cls.count)
t1=Test()
t2=Test()
t3=Test()
t4=Test()
t5=Test()
Test.noOfObjects()
```

```
The number of objects created for testclass: 5
>>>
```

3. Static Methods:

- In general these methods are general utility methods.
- Inside these methods we won't use any instance or class variables.
- Here we won't provide self or cls arguments at the time of declaration.
- We can declare static method explicitly by using @staticmethod decorator
- We can access static methods by using classname or object reference
- Note: In general we can use only instance and static methods.
- Inside static method we can access class level variables by using class name.
- class methods are most rarely used methods in python.

```
class ITC:
    @staticmethod
    def add(x,y):
        print('TheSum:',x+y)
    @staticmethod
    def product(x,y):
        print('TheProduct:',x*y)
    @staticmethod
    def average(x,y):
        print('Theaverage:',(x+y)/2)
ITC.add(10,20)
ITC.product(10,20)
ITC.average(10,20)
```

```
TheSum: 30
TheProduct: 200
Theaverage: 15.0
>>>
```

classmethod() is considered un-Pythonic so in newer Python versions, you can use the @classmethod decorator for classmethod definition.

- The difference between a static method and a class method is:
- Static method knows nothing about the class and just deals with the parameters
- Class method works with the class since its parameter is always the class itself.
- the class method is always attached to a class with first argument as the class itself *cls*.

def classmethod(cls, arg)

Python In-built class functions

The in-built functions defined in the class are described in the following table.

SN	Function	Description
1	getattr(obj,name,default)	It is used to access the attribute of the object.
2	setattr(obj, name,value)	It is used to set a particular value to the specific attribute of an object.
3	delattr(obj, name)	It is used to delete a specific attribute.
4	hasattr(obj, name)	It returns true if the object contains some specific attribute.

Built-in class attributes

Along with the other attributes, a python class also contains some built-in class attributes which provide information about the class.

The built-in class attributes are given in the below table.

SN	Attribute	Description
1	__dict__	It provides the dictionary containing the information about the class namespace.
2	__doc__	It contains a string which has the class documentation
3	__name__	It is used to access the class name.
4	__module__	It is used to access the module in which, this class is defined.
5	__bases__	It contains a tuple including all base classes.

Python Inheritance:

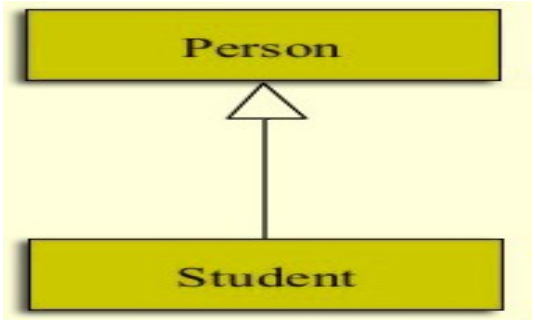
- Inheritance is an important aspect of the object-oriented paradigm.
- Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.
- In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class.
- A child class can also provide its specific implementation to the functions of the parent class.
- In python, **a derived class can inherit base class by just mentioning the base in the bracket after the derived class name.**
- Consider the following syntax to inherit a base class into the derived class.

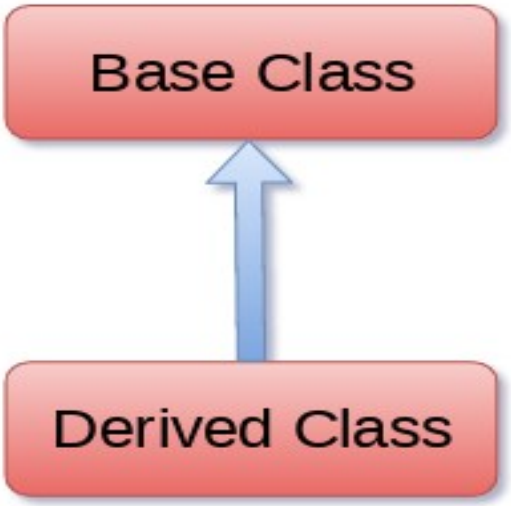
Person can be called any of the following:

Super Class or Parent Class or Base Class

Likewise, Student here is:

Sub Class or Child Class or Derived Class



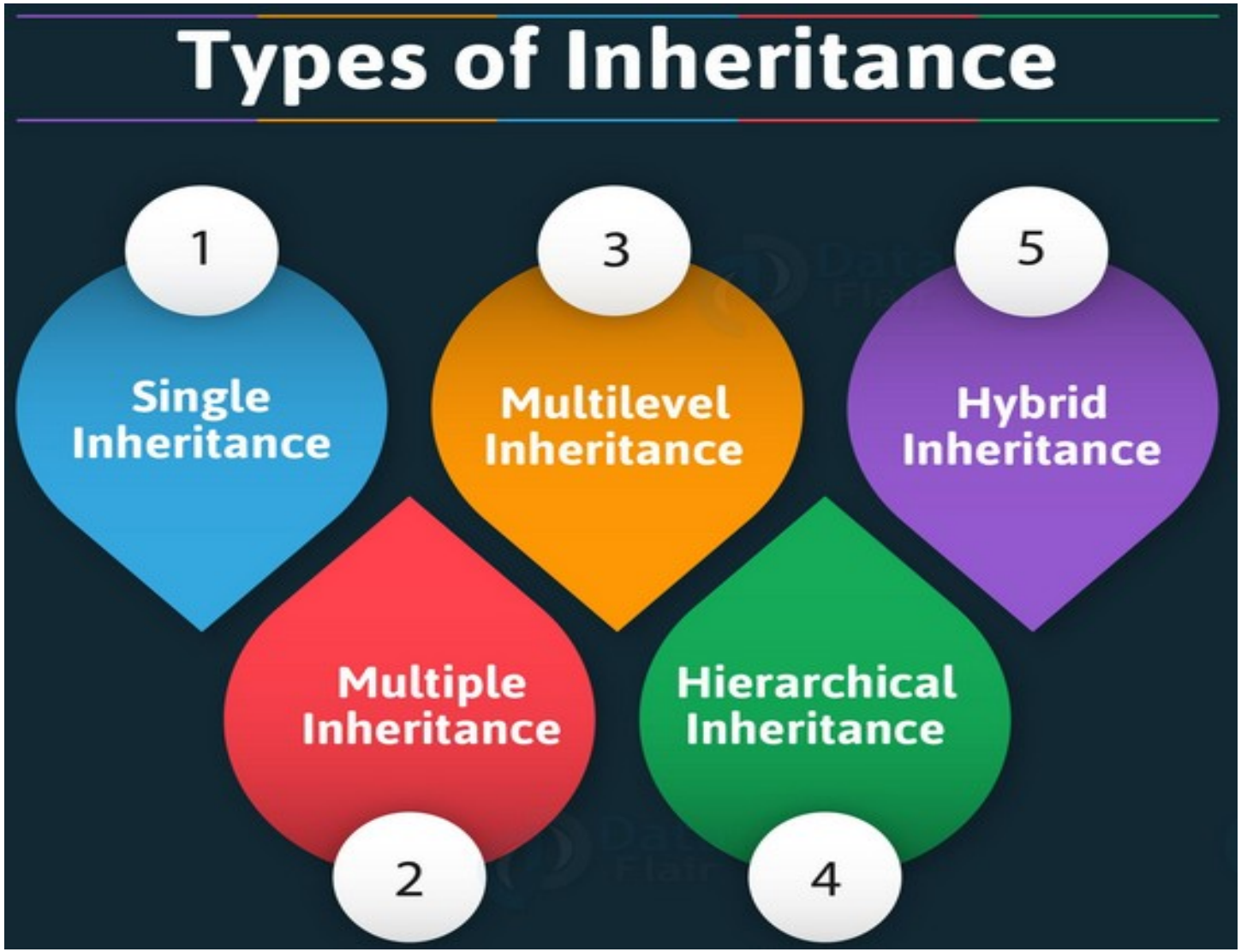


Syntax

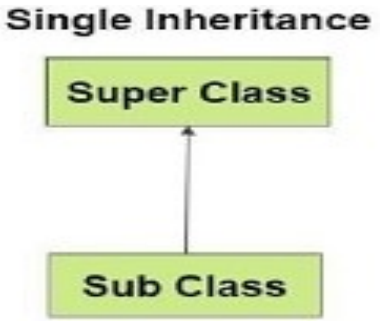
```
class derived-class(base class):
    <class-suite>
```

Example:

```
class A:
# variable of class A
# functions of class A
class B(A):
 # variable of class A
# functions of class A
# add more properties to class B
```


Types of Inheritance
1
Single Inheritance
3
Multilevel Inheritance
5
Hybrid Inheritance
Multiple Inheritance
2
Hierarchical Inheritance
4

1. Single inheritance: When a child class inherits from only one parent class, it is called as single inheritance.



Syntax

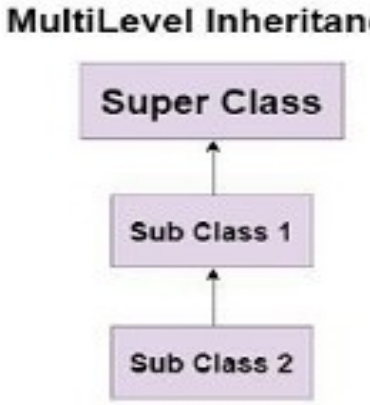
```
class derived-class(base class):
    <class-suite>
```

```
class A:
    def __init__(self):
        self.a=int(input("enter a"))
        self.b=int(input("enter b"))
class B(A):
    def add(self):
        print("sum=",self.a+self.b)
b=B()
b.add()
```

```
enter a4
enter b5
sum= 9
```

2. **Multi-Level inheritance:** when a derived class inherits from another derived class.

There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.



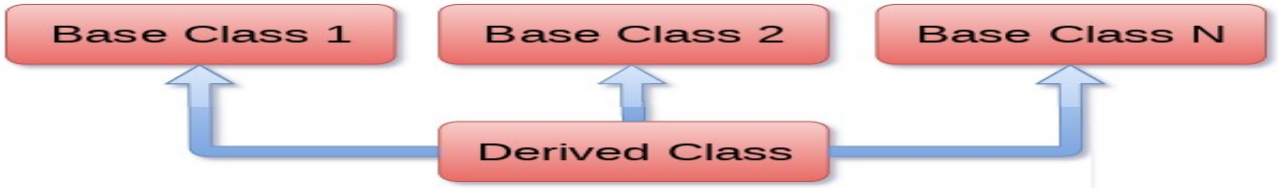
Syntax

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
.
```

```
class Sq:
    def readl(self):
        self.l=int(input("enter l"))
    def areaofsq(self):
        print("area of square=",self.l*self.l)

class Rect(Sq):
    def readb(self):
        self.b=int(input("enter b"))
    def areaofrect(self):
        print("area of rectangle=",self.l*self.b)
class Box(Rect):
    def readh(self):
        self.h=int(input("enter h"))
    def volofbox(self):
        print("volume=",self.l*self.b*self.h)
b=Box()
b.readl()
b.areaofsq()
b.readb()
b.areaofrect()
b.readh()
b.volofbox()
```

3.Multiple inheritance:

- Python provides us the flexibility to inherit multiple base classes in the child class.

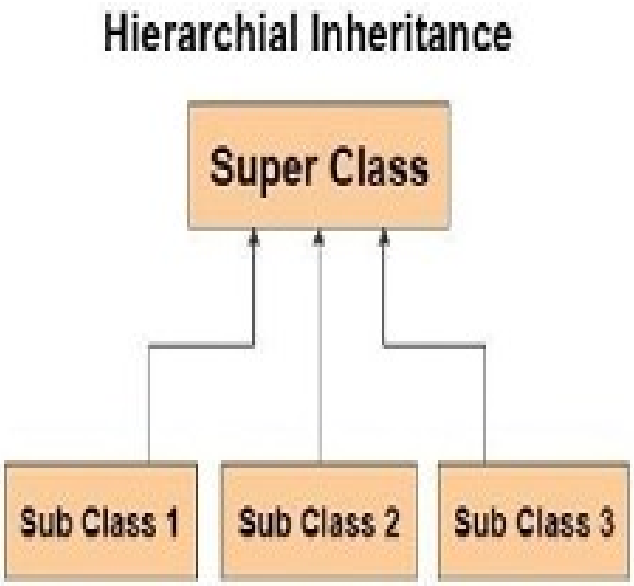


Syntax

```
class Base1:
    <class-suite>

class Base2:
    <class-suite>
.
.
.
class BaseN:
    <class-suite>

class Derived(Base1, Base2, ..... BaseN):
    <class-suite>
```

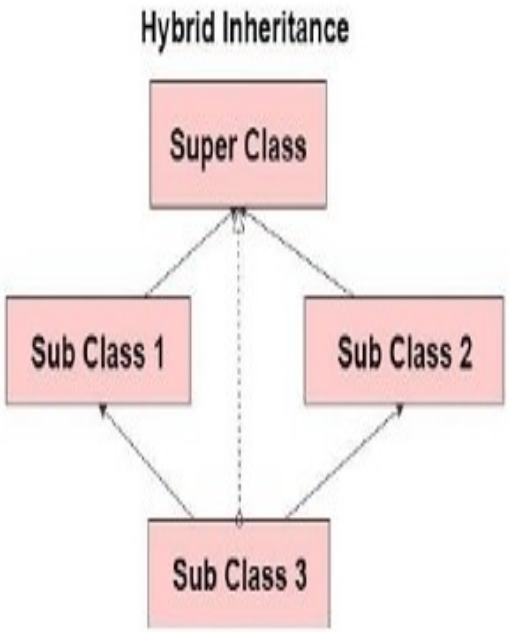
```
class A:
    def reada(self):
        self.a=int(input("enter a"))

class B:
    def readb(self):
        self.b=int(input("enter b"))
class C(A,B):
    def sub(self):
        print("sum=",self.a-self.b)
c1=C()
c1.reada()
c1.readb()
c1.sub()
```

Hierarchical inheritance More than one derived classes are created from a single base.




```
class Demo:
    def __init__(self):
        self.a=int(input("enter a"))
        self.b=int(input("enter b"))

class A(Demo):
    def add(self):
        print("sum=",self.a+self.b)
class B(Demo):
    def sub(self):
        print("sum=",self.a-self.b)
class C(Demo):
    def mul(self):
        print("sum=",self.a*self.b)
a=A()
a.add()
b=B()
b.sub()
c=C()
c.mul()
```

Hybrid inheritance: is combination of more than one form of inheritance. Basically, it is a blend of more than one type of inheritance.




```
class Snist:
    def fun1(self):
        print("I am in snist")
class Mech(Snist):
    def fun2(self):
        print(" I am in Mech")
class MechA(Mech):
    def fun3(self):
        print("I am from Section A")
class MechB(Mech):
    def fun4(self):
        print("I am from Section B")
A=MechA()
A.fun1()
A.fun2()
A.fun3()
B=MechB()
B.fun1()
B.fun2()
B.fun4()
```

Polymorphism:

- Poly means many. Morphs means forms.Polymorphism means 'Many Forms'.
- variable, object or a method exhibits different behavior in different contexts is called **polymorphism**

Eg1:+ operator acts as concatenation and arithmetic addition

```
print(10+20)#30
print('Sree'+'nidhi')#Sreenidhi
```

Eg2:* operator acts as multiplication and repetition operator

```
`print(10*20)#200
print('snist'*3)#snistsnistsnist
```

Eg3:We can use deposit() method to deposit cash or cheque or dd

```
deposit(cash)
deposit(cheque)
deposit(dd)
```

1.Operator Overloading

2.Method Overloading

3.Method overriding

Operator overloading:We can use the same operator for multiple purposes, which is nothing but operator overloading.

- * operator can be used for multiplication and string repetition purposes.
- Python supports operator overloading.+ operator can be used for Arithmetic addition ,complex addition , String concatenation and combine two lists.But + operator can not add two objects.
- We can overload +,*,-…etc operators to act upon objects using special methods

The following is the list of operators and corresponding magic methods.

```
+   ---> object.__add__(self,other)
-   ---> object.__sub__(self,other)
*   ---> object.__mul__(self,other)
/   ---> object.__div__(self,other)
//   ---> object.__floordiv__(self,other)
%   ---> object.__mod__(self,other)
**   ---> object.__pow__(self,other)
+=   ---> object.__iadd__(self,other)
-=   ---> object.__isub__(self,other)
*=   ---> object.__imul__(self,other)
/=   ---> object.__idiv__(self,other)
//=   ---> object.__ifloordiv__(self,other)
%=   ---> object.__imod__(self,other)
**=   ---> object.__ipow__(self,other)
<   ---> object.__lt__(self,other)
<=   ---> object.__le__(self,other)
>   ---> object.__gt__(self,other)
>=   ---> object.__ge__(self,other)
==   ---> object.__eq__(self,other)
!=   ---> object.__ne__(self,other)
```

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

b1=Book(100)
b2=Book(200)
print(b1+b2)
```

```
D:\durga_classes>py test.py
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    print(b1+b2)
TypeError: unsupported operand type(s) for +: 'Book' and 'Book'
```

- We can overload + operator to work with Book objects also. i.e
- Python supports Operator Overloading.
- 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.

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

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

b1=Book(100)
b2=Book(200)
print('The Total Number of Pages:',b1+b2)
```

Output: The Total Number of Pages: 300

```
#Overloading > and <= operators for Student class objects
class Student:
    def __init__(self,name,marks):
        self.name=name
        self.marks=marks
    def __gt__(self,other):
        return self.marks>other.marks
    def __le__(self,other):
        return self.marks<=other.marks

print("10>20=",10>20)
s1=Student("Ramesh",100)
s2=Student("Suresh",200)
print("s1>s2=",s1>s2)
print("s1<s2=",s1<s2)
print("s1<=s2=",s1<=s2)
print("s1>=s2=",s1>=s2)
```

```
10>20= False
s1>s2= False
s1<s2= True
s1<=s2= True
s1>=s2= False
>>>
```

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

```
This Month Salary: 12500
>>>
```

Method Overloading:

Defining multiple methods having same name but different type of arguments then those methods are said to be overloaded methods.

```
class X:
        def product(self,a,b):
                print(a*b)
        def  product(self,a,b,c):
                print(a*b*c)
X1=X()
X1.product(4,5)   # produces an error
```

- 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.
- But we can make the same function work differently using **Default arguments and Variable length arguments**
- **In python, If single method can perform more than one task is called method overloading**

```
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)
```

Output: two-arg method

```
class addition
        def
    add(self,x=10,y=20)
                print(a+b)
a1=addition()
a1.add()
a1.add(20)
a1.add(30,40)
a1.add("hello","snist")
a1.add(23.5,56.2)
a1.add([11,12],[34,45])
```

```
#Demo Program with Variable Number of Arguments:
class Test:
    def sum(self,*a):
        total=0
        for x in a:
            total=total+x
        print('TheSum:',total)
t=Test()
t.sum(10,20)
t.sum(10,20,30)
t.sum(10)
t.sum()
```

```
TheSum: 30
TheSum: 60
TheSum: 10
TheSum: 0
>>>
```

Constructor Overloading:

- Constructor overloading is not possible in Python.
- If we define multiple constructors then the last constructor will be considered.

```
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')
#t1=Test()
#t1=Test(10)
t1=Test(10,20)
```

Output: Two-Arg constructor

- In the above program only Two-Arg Constructor is available.
- But based on our requirement we can declare constructor with default arguments and variable number of arguments.

```
#Constructor with Default Arguments:
class Test:
    def __init__(self,a=None,b=None,c=None):
        print('Constructor with 0|1|2|3 number of arguments')


t1=Test()
t2=Test(10)
t3=Test(10,20)
t4=Test(10,20,30)
```

```
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
>>>
```

```
#Constructor with Variable Number of Arguments:
class Test:
    def __init__(self,*a):
        print('Constructor with variable number of arguments')


t1=Test()
t2=Test(10)
t3=Test(10,20)
t4=Test(10,20,30)
t5=Test(10,20,30,40,50,60)
```

```
Constructor with variable number of arguments
Constructor with variable number of arguments
Constructor with variable number of arguments
Constructor with variable number of arguments
Constructor with variable number of arguments
>>>
```

Method overriding: defining same method with same number of arguments in both super class and sub class is called method overriding

- What ever members available in the parent class 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.

```
#Demo Program for Method overriding
class P:
    def property(self):
        print('Gold+Land+Cash+Power')
    def marry(self):
        print('Ram')
class C(P):
    def marry(self):
        print('Krish')

c=C()
c.property()
c.marry()
```

```
Gold+Land+Cash+Power
Krish
>>>
```

Super() method

- If we write a method or constructor in the sub class with exactly same name as that of super class method or constructor, it will override super class method or constructor
- super class method and constructor are not available to the sub class, only sub class method and constructor are accessible from the sub class object.
- Super() is a built in method which is useful to call the super class constructor or methods from the base class.

```
super().__init__()                  #call super class default constructor
super().__init__(arguments)         #call super class parameterized constructor
super().method()                    #call super class method
```

We can use super class name to call super class constructor or method

```
super class name.__init__(self)                    #call super class default constructor
super class name. __init__(self,argument) #call super class parameterized constructor
super class name.method(self)                      #call super class  method
```

- **From Overriding method of child class, we can call parent class method also by using super() method.**

```
class P:
    def property(self):
        print('Gold+Land+Cash+Power')
    def marry(self):
        print('Ram')
class C(P):
    def marry(self):
        super().marry()
        print('Krish')

c=C()
c.property()
c.marry()
```

```
Gold+Land+Cash+Power
Ram
Krish
>>>
```

```
class  square:
    def __init__(self):
    self.a=int(input("enter side"))
    def area(self):
    print("area of square=",self.a**2)
class  rectangle(square):
    def __init__(self):
    super().__init__()
    self.b=int(input("enter other side"))
    def area(self):
    super().area()
    print("area of rectangle",self.a*self.b)
r=rectangle()
r.area()
```

Program for Constructor overriding:

```
class P:
    def __init__(self):
        print('Parent Constructor')

class C(P):
    def __init__(self):
        print('Child Constructor')

c=C()
```

Output: Child Constructor

- In the above example, if child class does not contain constructor then parent class constructor will be executed From child class constructor we can call parent class constructor by using super() method.

```python
#Program to call Parent class constructor by using super()
class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age
class Employee(Person):
    def __init__(self,name,age,eno,esal):
        super().__init__(name,age)
        self.eno=eno
        self.esal=esal

    def display(self):
        print('Employee Name:',self.name)
        print('Employee Age:',self.age)
        print('Employee Number:',self.eno)
        print('Employee Salary:',self.esal)
        print('\n')


e1=Employee('Ram',25,1213,36000)
e1.display()
e2=Employee('Sam',28,872426,39000)
e2.display()
```

```
Employee Name: Ram
Employee Age: 25
Employee Number: 1213
Employee Salary: 36000


Employee Name: Sam
Employee Age: 28
Employee Number: 872426
Employee Salary: 39000
```

Data hiding

- Abstraction is an important aspect of object-oriented programming.
- In python, we can also perform data hiding by adding the double underscore (___) as a prefix to the attribute which is to be hidden.
- After this, the attribute will not be visible outside of the class through the object.

Ex

```
class  My class:
        def  __init__(self):
                self.__y=3
m=Myclass()
print(m.y)  # produces an error ie; Attribute Error
```

Regular expressions

- A Regular Expression is a string that contains symbols and characters to find and extract the information needed by us. A Regular expression helps us to search information, match, find and split information as per our requirements.
- It is also called regex.
- The Regular Expressions are used to perform the following operations.

  Matching strings
  Searching for strings
  Finding all strings
  Splitting a string into pieces
  Replacing strings

- python provides re module, it contains the methods like compile(), search(), match(), findall(), split() etc are used in finding the information in the available data.so when we write regular expressions, we should import re module
- The string is prefixed with r or R is called **raw string**

```
str=r"hello \n snist"
print(str)                    #hello \n snist
```

- If regular expressions should written as raw strings, the normal meaning of escape characters(\n,\t,\v,\b,\'…) are escaped .The escape characters interpreted as special characters in the regular expression.

```
pat=r'\b a\w* \b              \\ raw string
```

- If we do not want to write the regular expressions as raw strings, then use another backslash before escape characters.

```
pat='\\b a\w* \\b             \\ normal string
```

MetaCharacters: metacharacters are used to specify regular expressions
Metacharacters are characters that are interpreted in a special way by a RegEx engine. Here's a list of metacharacters: * + ? . [] ^ $ {} () \ |

Quantifiers

Character	Description
*	Zero or more occurrences of the preceding expression
+	one or more repetitions of the preceding expression
?	Zero or one repetition of the preceding expression
{m}	Exactly m occurences
{m,n}	from m to n occurrences(m defaults to 0, n to infinity)
{m,}	At least m occurrences

Special characters

.	Matches any character except new line
[…]	set of possible characters.
[^…]	matches every character except the ones inside brackets
^	check if a string **starts with** a certain character or word
$	check if a string **ends with** a certain character or word
\	escape special characters nature
(…)	matches the regular expression inside the parenthesis
R|S	matches either regex R or regex S

- We can develop Regular Expression Based applications by using python module: **re**
- This module contains several inbuilt functions to use Regular Expressions very easily in our applications.

compile()

- re module contains compile() function to compile a pattern into RegexObject.(Python Object)
- **pattern = re.compile("ab")**

```
import re
pattern =re.compile('Python')
print(type(pattern))
```

```
<class '_sre.SRE_Pattern'>
>>>
```

finditer():

- Returns an Iterator object which yields Match object for every Match.
- We can check how many matches are available.

matcher = pattern.finditer("Python is very is easy")

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**

```
import re
count=0
pattern =re.compile('ab')
matcher=pattern.finditer('abaabab')
for match in matcher:
    count+=1
    print('match is available at start index:',match.start())
    print(match.start(),"---",match.end(),"---",match.group())
print('Number of accurences:',count)
```

```
match is available at start index: 0
0 --- 2 --- ab
match is available at start index: 3
3 --- 5 --- ab
match is available at start index: 5
5 --- 7 --- ab
Number of accurences: 3
>>>
```

```
import re
count=0
matcher=re.finditer("ab","abaababa")
for match in matcher:
    count+=1
    print('match is available at start index:',match.start())
    print(match.start(),"---",match.end(),"---",match.group())
print('Number of accurences:',count)
```

```
match is available at start index: 0
0 --- 2 --- ab
match is available at start index: 3
3 --- 5 --- ab
match is available at start index: 5
5 --- 7 --- ab
Number of accurences: 3
>>>
```

Quantifiers:

- We can use quantifiers to specify the **number of occurrences to match.**
- a→ Exactly one 'a‘
- A+→ Atleast one 'a‘
- a*→Any number of a's including zero number
- a?→Atmost 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

```
import re
matcher=re.finditer("x","abaabaaab")
for match in matcher:
    print(match.start(),"......",match.group())
```

```
import re
str1='abaabaaab'
res=re.findall('a',str1)
print(res)
```

x = a:

0 ..... a
2 ..... a
3 ..... a
5 ..... a
6 ..... a
7 ..... a

x = a+:

0 ..... a
2 ..... aa
5 ..... aaa

x = a*:

0 ..... a
1 .....
2 ..... aa
4 .....
5 ..... aaa
8 .....
9 .....

x = a?:

0 ..... a
1 .....
2 ..... a
3 ..... a
4 .....
5 ..... a
6 ..... a
7 ..... a
8 .....
9 .....

x = a{3}:

5 ..... aaa

x = a{2,4}:

2 ..... aa
5 ..... aaa

Character classes:

We can use character classes to search a group of characters

1. [abc]===>Either a or b or c
2. [^abc] ===>Except a and b and c
3. [a-z]==>Any Lower case alphabet symbol
4. [A-Z]===>Any upper case alphabet symbol
5. [a-zA-Z]==>Any alphabet symbol
6. [0-9] Any digit from 0 to 9
7. [a-zA-Z0-9]==>Any alphanumeric character
8. [^a-zA-Z0-9]==>Except alphanumeric characters(Special Characters)

Eg:

```
import re
matcher=re.finditer("x","a7b@k9z")
for match in matcher:
    print(match.start(),"......",match.group())
```

```
import re
str1='a7b@k9z'
res=re.findall('.',str1)
print(res)
```

x = [abc]

0 ..... a
2 ..... b

x = [^abc]

1 7
3 ..... @
4 ..... k
5 9
6 ..... z

x = [a-z]

0 ..... a
2 ..... b
4 ..... k
6 ..... z

x = [0-9]

1 7
5 9

x = [a-zA-Z0-9]

0 ..... a
1 7
2 ..... b
4 ..... k
5 9
6 ..... z

x = [^a-zA-Z0-9]

3 ..... @

x = .

0 ..... a
1 7
2 ..... b
3 .....

Note:

^x ➔ It will check whether target string starts with x or not

x$ ➔ It will check whether target string ends with x or not

```
import re
matcher=re.finditer("^a","abaabaaab")
for match in matcher:
    print(match.start(),"......",match.group())
```

```
0 ...... a
>>>
```

```
import re
matcher=re.finditer("a$","abaabaaab")
for match in matcher:
    print(match.start(),"......",match.group())
```

```
>>>
>>>
```

Sequence characters in Regular Expressions

Character	Description
\d	represents any digit([0-9])
\D	represents any non digit([^0-9])
\s	represents white space[\t\n\r\f\v].)
\S	represents non whitespace [^\t\n\r\f\v].)
\w	represents any alphanumeric([A-Za-z0-9_])
\W	represents any non alphanumeric([^A-Za-z0-9_]
\b	represents a space around words
\A	matches only at start of the string
\z	matches only at end of the string

```
import re
matcher=re.finditer("x","a7b k@9z")
for match in matcher:
    print(match.start(),".....",match.group())
```

x = \s:

3 .....

x = \S:

0 ..... a
1 7
2 ..... b
4 ..... k
5 ..... @
6 9
7 ..... z

x = \d:

1 7
6 9

x = \D:

0 ..... a
2 ..... b
3 .....
4 ..... k
5 ..... @
7 ..... z

x = \w:

0 ..... a
1 7
2 ..... b
4 ..... k
6 9
7 ..... z

x = \W:

3 .....
5 ..... @

Important functions of re module:

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

1. 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.

```
import re
s=input("Enter pattern to check:")
m=re.match(s,"abcabdefg")
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")
```

```
Enter pattern to check:abc
Match is available at the beginning of the String
Start Index: 0 and End Index: 3
```

```
Enter pattern to check:cde
Match is not available at the beginning of the String
>>>
```

```
import re
s=input("Enter pattern to check:")
m=re.match(s,"prem")
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")
```

```
Enter pattern to check:pre
Match is available at the beginning of the String
Start Index: 0 and End Index: 3
>>> ========== RESTART ==========
>>>
Enter pattern to check:pem
Match is not available at the beginning of the String
>>> ========== RESTART ==========
>>>
Enter pattern to check:premnadh
Match is not available at the beginning of the String
>>>
```

2. 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

3. findall():

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

```
import re
l=re.findall("[0-9]","a7b9c7wxhji835")
print(l)
```

```
['7', '9', '7', '8', '3', '5']
>>>
```

```
import re
s=input("Enter pattern to check:")
m=re.search(s,"sreenidhi")
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")
```

```
Enter pattern to check:sre
Match is available
First Occurrence of match with start index: 0 and end index: 3
>>> ========== RESTART ==========
>>>
Enter pattern to check:srin
Match is not available
>>> ========== RESTART ==========
>>>
Enter pattern to check:nidhi
Match is available
First Occurrence of match with start index: 4 and end index: 9
>>> ========== RESTART ==========
>>>
Enter pattern to check:sreei
Match is not available
>>> ========== RESTART ==========
>>>
Enter pattern to check:sreenidhi
Match is available
First Occurrence of match with start index: 0 and end index: 9
>>>
```

4. sub():

- sub means substitution or replacement

re.sub(regex,replacement,target string)

- In the target string every matched pattern will be replaced with provided replacement.

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

Output: #7#9#5#8#

Every alphabet symbol is replaced with # symbol

5. 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.

(resultstring, number of replacements)

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

Output:

```
D:\python_classes>py test.py
('#7#9#5#8#', 5)
The Result String: #7#9#5#8#
The number of replacements: 5
```

6.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.

```
import re
l=re.split(",","sunny,bunny,chinny,vinny,pinny")
print(l)
for t in l:
    print(t)
```

Output:

```
D:\python_classes>py test.py
['sunny', 'bunny', 'chinny', 'vinny', 'pinny']
sunny
bunny
chinny
vinny
pinny
```

```
import re
l=re.split("\.","SNIST.IT.SECTION-C")
for t in l:
    print(t)
```

```
SNIST
IT
SECTION-C
>>>
```

7. fullmatch():

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

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

```
Enter pattern to check:snist
Full String Matched
>>> ==========
>>>
Enter pattern to check:sni
Full String not Matched
>>> ==========
>>>
Enter pattern to check:snists
Full String not Matched
>>>
```

8. finditer():

- Returns the Iterator yielding a match object for each match.
- On each match object we can call start(),end() and group() functions.

```
import re
itr=re.finditer("[a-z]","3nist")
for m in itr:
    print(m.start(),"...",m.end(),"...",m.group())
```

```
0 ... 1 ... s
1 ... 2 ... n
2 ... 3 ... i
3 ... 4 ... s
4 ... 5 ... t
>>>
```

Matching Versus Searching

- Python offers two different primitive operations based on regular expressions: **match** checks for a match only at the beginning of the string, while **search** checks for a match anywhere in the string

^ symbol:

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

Eg:

res=re.search("^Learn",s)

if the target string starts with Learn then it will return Match object,otherwise returns None.

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

Output: Target String starts with Learn

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

Eg: res=re.search("Easy$",s)

If the target string ends with Easy then it will return Match object,otherwise returns None.

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

Output: Target String ends with Easy

Note: If we want to ignore case then we have to pass 3rd argument re.IGNORECASE for search() function.

Eg: res = re.search("easy$",s,re.IGNORECASE)

```
import re
s="Learning Python is Very Easy"
res=re.search("easy$",s,re.IGNORECASE)
if res != None:
    print("Target String ends with Easy by ignoring case")
else:
    print("Target String Not ends with Easy by ignoring case")
```

Output: Target String ends with Easy by ignoring case

Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these −

Flags	Description
re.I	Performs case-insensitive matching.
re.L	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B).
re.M	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
re.S	Makes a period (dot) match any character, including a newline.
re.U	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
re.X	Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.