Polymorphism

“Poly” means “Many” and “Morphism” is “forms”
Defining one thing in many forms is called polymorphism.
In python polymorphism is implemented using

1. Method Overriding
2. Operator Overloading

Operator Overloading

Operator overloading in Python refers to the ability to change the behavior of operators based on the operands that they act upon.

This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.

For every operator python provides a special method/function which called operator function.

For every operator python provides a magic method. It is called operator magic method, this method executed automatically.

If python programmer wants provide an operator for user defined data type, operator magic method must be overload.

Python provides the following operator magic methods.

Operator	Expression	Magic Method
+	A + B	__add__(self, other)
-	A – B	__sub__(self, other)
*	A * B	__mul__(self, other)
/	A / B	__truediv__(self, other)
//	A // B	__floordiv__(self, other)
%	A % B	__mod__(self, other)
**	A ** B	__pow__(self, other)
>>	A >> B	__rshift__(self, other)
<<	A << B	__lshift__(self, other)
&	A & B	__and__(self, other)
|	A | B	__or__(self, other)
^	A ^ B	__xor__(self, other)
<	A < B	__LT__(SELF, OTHER)
>	A > B	__GT__(SELF, OTHER)
<=	A <= B	__LE__(SELF, OTHER)
>=	A >= B	__GE__(SELF, OTHER)
==	A == B	__EQ__(SELF, OTHER)
!=	A != B	__NE__(SELF, OTHER)
-=	A - = B	__ISUB__(SELF, OTHER)
+=	A += B	__IADD__(SELF, OTHER)
*=	A *= B	__IMUL__(SELF, OTHER)
/=	A /= B	__IDIV__(SELF, OTHER)
//=	A //= B	__IFLOORDIV__(SELF, OTHER)
%=	A %= B	__IMOD__(SELF, OTHER)
**=	A **= B	__IPOW__(SELF, OTHER)
>>=	A >>= B	__IRSHIFT__(SELF, OTHER)
<<=	A <<= B	__ILSHIFT__(SELF, OTHER)
&=	A &= B	__IAND__(SELF, OTHER)

Example:

```
class Point:
    def __init__(self,p,q):
        self.__x=p
        self.__y=q
    def __add__(self,other):
        p3=Point(0,0)
        p3.__x=self.__x+other.__x
        p3.__y=self.__y+other.__y
        return p3
    def __str__(self):
        return f'{self.__x},{self.__y}'



point1=Point(100,200)
point2=Point(50,60)
point3=point1+point2 # point1.__add__(point2)

print(point1)
print(point2)
print(point3)

list1=[10,20,30]
list2=[40,50,60]
list3=list1+list2
print(list1,list2,list3,sep="\n")
```

Output

```
100,200
50,60
150,260
[10, 20, 30]
[40, 50, 60]
[10, 20, 30, 40, 50, 60]
```

Example:

```
class Marks:
```

```
    def __init__(self,s1,s2):
        self.__sub1=s1
        self.__sub2=s2
    def __eq__(self,other):
        if self.__sub1==other.__sub1 and
self.__sub2==other.__sub2:
            return True
        else:
            return False


stud1=Marks(50,60)
stud2=Marks(50,60)
print(stud1==stud2)
print(id(stud1),id(stud2))
```

Output

```
True
2680866421888 2680866421936
```

Example:

```
class Complex:
    def __init__(self):
        self.__real=0.0
        self.__img=0.0
    def set_real(self,r):
        self.__real=r
    def set_img(self,i):
        self.__img=i
    def get_real(self):
        return self.__real
    def get_img(self):
        return self.__img
    def __add__(self,other):
        c3=Complex()
        c3.__real=self.__real+other.__real
        c3.__img=self.__img+other.__img
        return c3
```

```
comp1=Complex()
print(comp1.get_real(),comp1.get_img())
comp1.set_real(1.5)
comp1.set_img(2.5)
print(comp1.get_real(),comp1.get_img())
comp2=Complex()
comp2.set_real(1.2)
comp2.set_img(1.5)
print(comp2.get_real(),comp2.get_img())
comp3=comp1+comp2
print(comp3.get_real(),comp3.get_img())
```

Output

```
0.0 0.0
1.5 2.5
1.2 1.5
2.7 4.0
```

Abstract classes and abstract methods (abc module)

“abc” is called abstract base class module. It is a predefined module which comes with python software. This module is used for implementation of abstract classes and abstract methods.

What is abstract class?

Abstract class is collection of abstract methods, non abstract methods and variables.
Abstract class is an abstract data type, which allows to build similar data types.
Abstract class defines set of rules and regulations which has to implemented or followed by every derived class.
Abstract class is used for inheriting purpose and this class cannot used for creating object.

Syntax:

```
class <abstract-class-name>(abc.ABC):
        variables
        abstract methods
        non abstract methods
```

any class inherited “ABC” class of “abc” module is called abstract base class.

Abstract method

An empty method is called abstract method (OR) a method inside class without implementation is called abstract method.
Abstract method is defined inside abstract class
Abstract method defines a protocol or rule, which has to be implemented by every sub class/derived class.
Abstract method must override by derived class.

Syntax:
@abc.abstractmethod
def <method-name>(self,param,param,param,..):
pass

The class which inherits abstract class must override abstract methods.

Example:

```
import abc

class Animal(abc.ABC):
    @abc.abstractmethod
    def sleep(self):
        pass

class Dog(Animal):
    def sleep(self):
        print("Dog Sleep Day Time...")

class Cat(Animal):
    def sleep(self):
        print("Cat Sleep in Night Time...")
```

```
# a=Animal() Error
dog1=Dog()
cat1=Cat()

dog1.sleep()
cat1.sleep()
```

Output

```
Dog Sleep Day Time...
Cat Sleep in Night Time...
```

Example:

```
import abc

class A(abc.ABC):
    @abc.abstractmethod
    def m1(self):
        pass
    def m2(self):
        print("m2 of A class")

class B(A):
    def m1(self):
        print("overriding method")
    def m3(self):
        print("m3 of B class")

objb=B()
objb.m3()
objb.m1()
objb.m2()
```

Output

```
m3 of B class
overriding method
m2 of A class
```

The class which inherits abstract class and provides implementations of abstract methods is called concrete class.