OOP in Python

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

In Python, the concept of OOP follows some basic principles:

Inheritance: A process of using details from a new class without modifying existing class.

Encapsulation: Hiding the private details of a class from other objects.

Polymorphism: A concept of using common operation in different ways for different data input.

Class,Object,Methods

Class:

A class is a blueprint for the object.

```
class Python:
    pass
```

Class,Object,Methods

Object:

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

```
obj = Python()
```

Creating Class and Object in Python

```
class Point:                    #naming convention
    def hello(self):       #method
        print("hello")
point1 = Point()                 #creating an object
point1.hello()
```

Methods:

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Self parameter

It is a reference to the current instance of the class.

It has to be the first parameter of any function in the class.

It is used to access variables that belongs to the class.

We can call it whatever we like (not only self).

__init__() Function

- It is always executed when the class is being initiated.
- The function to assign values to object properties.

```
class Point:
   def __init__(self,x,y):          #self = reference to the current object (This
            is called constructor)
      self.x = x                     #setting the x attribute to the x argument passed to this function


      self.y = y
   def hello(self):            #method
      pass
point1 = Point(10,20)          #creating an object
print(point1.x)
print(point1.y)
```

Inheritance

```
class Profile:
  def __init__(self, name, address):
    self.name = name
    self.address = address

class Hr(Profile):
     def info(self):
          print(self.name, self.address)

x = Hr(“Ram”, “Nepal”)
x.info()
```

__str__ method

This method returns the string representation of the object. This method is called when print() or str() function is invoked on an object.

__repr__

Python __repr__() function returns the object representation. It could be any valid python expression such as tuple, dictionary, string etc.

__str__ method python

```
class Person:
    name = ""
    age = 0
    def __init__(self, personName,
personAge):
        self.name = personName
        self.age = personAge
def __str__(self):
        return
“Person(name='+self.name+',
age='+str(self.age)+ ')”
```

```
p = Person('Pankaj', 34)


# __str__() example
print(p)
print(p.__str__())

s = str(p)
print(s)
```

Python __repr__()

```
class Person:
    name = ""
    age = 0

    def __init__(self, personName, 
personAge):
        self.name = personName
        self.age = personAge

    def __repr__(self):
        return {‘name’:self.name, 
‘age’:str(self.age)}
```

```
p = Person('Pankaj', 34)


# __repr__() example
print(p.__repr__())
print(type(p.__repr__()))
print(repr(p))
```

Operator Overloading

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

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
p1 = Point(2,3)
p2 = Point(-1,2)
p1 + p2
```

Polymorphism in Python

```
class Parrot:
    def fly(self):
        print("Parrot can fly")
    def swim(self):
        print("Parrot can't swim")
class Penguin:
    def fly(self):
        print("Penguin can't fly")
    def swim(self):
        print("Penguin can swim")
# common interface
     def flying_test(bird):
          bird.fly()
```

```
#instantiate objects
blu = Parrot()
peggy = Penguin()
# passing the object
flying_test(blu)
flying_test(peggy)
```

Operator Overloading

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x,self.y)

    def __add__(self,other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x,y)
p1 = Point(2,3)
p2 = Point(-1,2)
print(p1 + p2)
```

Operator Overloading Special Functions in Python

Operator	Expression	Internally
Addition	p1 + p2	p1__add__(p2)
Subtraction	p1 - p2	p1__sub__(p2)
Multiplication	p1 * p2	p1__mul__(p2)
Power	p1 ** p2	p1__pow__(p2)
Division	p1 / p2	p1__truediv__(p2)
Floor Division	p1 // p2	p1__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1__mod__(p2)