**Unit-IV:**

Advance Python- OOPs concept: Class and object ,Attributes , Inheritance, Overloading, Overriding ,Data hiding .

Regular expressions---Match function ,Search function ,Matching VS Searching , Modifiers Patterns.

# 

# **Python Object Oriented Programming**

## **Introduction to OOPs in Python**

Python is a multi-paradigm programming language. Meaning, it supports different programming approach.

One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

* attributes
* behavior

Let's take an example:

Parrot is an object,

* name, age, color are attributes
* singing, dancing are behavior

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**

A class is a blueprint for the object.

We can think of class as an sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, parrot is an object.

The example for class of parrot can be :

class Parrot:  
 pass

Here, we use class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

## **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.

The example for object of parrot class can be:

obj = Parrot()

Here, obj is object of class Parrot.

Suppose we have details of parrot. Now, we are going to show how to build the class and objects of parrot.

#### **Example 1: Creating Class and Object in Python**

class Parrot:

# class attribute

species = "bird"

# instance attribute

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

# instantiate the Parrot class

blu = Parrot("Blu", 10)

woo = Parrot("Woo", 15)

# access the class attributes

print("Blu is a {}".format(blu.\_\_class\_\_.species))

print("Woo is also a {}".format(woo.\_\_class\_\_.species))

# access the instance attributes

print("{} is {} years old".format( blu.name, blu.age))

print("{} is {} years old".format( woo.name, woo.age))

When we run the program, the output will be:

Blu is a bird  
Woo is also a bird  
Blu is 10 years old  
Woo is 15 years old

In the above program, we create a class with name Parrot. Then, we define attributes. The attributes are a characteristic of an object.

Then, we create instances of the Parrot class. Here, blu and woo are references (value) to our new objects.

Then, we access the class attribute using \_\_class \_\_.species. Class attributes are same for all instances of a class. Similarly, we access the instance attributes using blu.name and blu.age. However, instance attributes are different for every instance of a class.

## **Methods**

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

### **Example 2 : Creating Methods in Python**

class Parrot:

# instance attributes

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

# instance method

def sing(self, song):

return "{} sings {}".format(self.name, song)

def dance(self):

return "{} is now dancing".format(self.name)

# instantiate the object

blu = Parrot("Blu", 10)

# call our instance methods

print(blu.sing("'Happy'"))

print(blu.dance())

When we run program, the output will be:

Blu sings 'Happy'  
Blu is now dancing

In the above program, we define two methods i.e sing() and dance(). These are called instance method because they are called on an instance object i.e blu.

## **Inheritance**

Inheritance is a way of creating new class for using details of existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

### **Example 3: Use of Inheritance in Python**

# parent class

class Bird:

def \_\_init\_\_(self):

print("Bird is ready")

def whoisThis(self):

print("Bird")

def swim(self):

print("Swim faster")

# child class

class Penguin(Bird):

def \_\_init\_\_(self):

# call super() function

super().\_\_init\_\_()

print("Penguin is ready")

def whoisThis(self):

print("Penguin")

def run(self):

print("Run faster")

peggy = Penguin()

peggy.whoisThis()

peggy.swim()

peggy.run()

When we run this program, the output will be:

Bird is ready  
Penguin is ready  
Penguin  
Swim faster  
Run faster

In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from swim()method. Again, the child class modified the behavior of parent class. We can see this from whoisThis() method. Furthermore, we extend the functions of parent class, by creating a new run() method.

Additionally, we use super() function before \_\_init\_\_() method. This is because we want to pull the content of \_\_init\_\_() method from the parent class into the child class.

## **Encapsulation**

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single “ \_ “ or double “ \_\_“.

### **Example 4: Data Encapsulation in Python**

**class Computer:**

**def \_\_init\_\_(self):**

**self.\_\_maxprice = 900**

**def sell(self):**

**print("Selling Price: {}".format(self.\_\_maxprice))**

**def setMaxPrice(self, price):**

**self.\_\_maxprice = price**

**c = Computer()**

**c.sell()**

**# change the price**

**c.\_\_maxprice = 1000**

**c.sell()**

**# using setter function**

**c.setMaxPrice(1000)**

**c.sell()**

When we run this program, the output will be:

Selling Price: 900  
Selling Price: 900  
Selling Price: 1000

In the above program, we defined a class Computer. We use \_\_init\_\_() method to store the maximum selling price of computer. We tried to modify the price. However, we can’t change it because Python treats the \_\_maxprice as private attributes. To change the value, we used a setter function i.e setMaxPrice() which takes price as parameter.

## **Polymorphism**

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).

Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism.

### **Example 5: Using 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)

When we run above program, the output will be:

Parrot can fly  
Penguin can't fly

In the above program, we defined two classes Parrot and Penguin. Each of them have common method fly() method. However, their functions are different. To allow polymorphism, we created common interface i.e flying\_test() function that can take any object. Then, we passed the objects blu and peggy in the flying\_test() function, it ran effectively.

# Python Objects and Class

## **What are classes and objects in Python?**

Python is an object oriented programming language. Unlike procedure oriented programming, where the main emphasis is on functions, object oriented programming stress on objects.

Object is simply a collection of data (variables) and methods (functions) that act on those data. And, class is a blueprint for the object.

We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.

As, many houses can be made from a description, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called **instantiation**.

## **Defining a Class in Python**

Like function definitions begin with the keyword [def](https://www.programiz.com/python-programming/keyword-list#def), in Python, we define a class using the keyword [class](https://www.programiz.com/python-programming/keyword-list#class).

The first string is called docstring and has a brief description about the class. Although not mandatory, this is recommended.

Here is a simple class definition.

class MyNewClass:  
 '''This is a docstring. I have created a new class'''  
 pass

A class creates a new local [namespace](https://www.programiz.com/python-programming/namespace) where all its attributes are defined. Attributes may be data or functions.

There are also special attributes in it that begins with double underscores (\_\_). For example, \_\_doc\_\_ gives us the docstring of that class.

As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

class MyClass:

"This is my second class"

a = 10

def func(self):

print('Hello')

# Output: 10

print(MyClass.a)

# Output: <function MyClass.func at 0x0000000003079BF8>

print(MyClass.func)

# Output: 'This is my second class'

print(MyClass.\_\_doc\_\_)

When you run the program, the output will be:

10  
<function 0x7feaa932eae8="" at="" myclass.func="">  
This is my second class

## **Creating an Object in Python**

We saw that the class object could be used to access different attributes.

It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a [function](https://www.programiz.com/python-programming/function) call.

>>> ob = MyClass()

This will create a new instance object named ob. We can access attributes of objects using the object name prefix.

Attributes may be data or method. Method of an object are corresponding functions of that class. Any function object that is a class attribute defines a method for objects of that class.

This means to say, since MyClass.func is a function object (attribute of class), ob.func will be a method object.

class MyClass:

"This is my second class"

a = 10

def func(self):

print('Hello')

# create a new MyClass

ob = MyClass()

# Output: <function MyClass.func at 0x000000000335B0D0>

print(MyClass.func)

# Output: <bound method MyClass.func of <\_\_main\_\_.MyClass object at 0x000000000332DEF0>>

print(ob.func)

# Calling function func()

# Output: Hello

ob.func()

You may have noticed the self parameter in function definition inside the class but, we called the method simply as ob.func() without any [arguments](https://www.programiz.com/python-programming/function-argument). It still worked.

This is because, whenever an object calls its method, the object itself is passed as the first argument. So, ob.func() translates into MyClass.func(ob).

In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

For these reasons, the first argument of the function in class must be the object itself. This is conventionally called self. It can be named otherwise but we highly recommend to follow the convention.

Now you must be familiar with class object, instance object, function object, method object and their differences.

## **Constructors in Python**

Class functions that begins with double underscore (\_\_) are called special functions as they have special meaning.

Of one particular interest is the \_\_init\_\_() function. This special function gets called whenever a new object of that class is instantiated.

This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

class ComplexNumber:

def \_\_init\_\_(self,r = 0,i = 0):

self.real = r

self.imag = i

def getData(self):

print("{0}+{1}j".format(self.real,self.imag))

# Create a new ComplexNumber object

c1 = ComplexNumber(2,3)

# Call getData() function

# Output: 2+3j

c1.getData()

# Create another ComplexNumber object

# and create a new attribute 'attr'

c2 = ComplexNumber(5)

c2.attr = 10

# Output: (5, 0, 10)

print((c2.real, c2.imag, c2.attr))

# but c1 object doesn't have attribute 'attr'

# AttributeError: 'ComplexNumber' object has no attribute 'attr'

c1.attr

In the above example, we define a new class to represent complex numbers. It has two functions, \_\_init\_\_() to initialize the variables (defaults to zero) and getData() to display the number properly.

An interesting thing to note in the above step is that attributes of an object can be created on the fly. We created a new attribute attr for object c2 and we read it as well. But this did not create that attribute for object c1.

## **Deleting Attributes and Objects**

Any attribute of an object can be deleted anytime, using the del statement. Try the following on the Python shell to see the output.

>>> c1 = ComplexNumber(2,3)  
>>> del c1.imag  
>>> c1.getData()  
Traceback (most recent call last):  
...  
AttributeError: 'ComplexNumber' object has no attribute 'imag'  
  
>>> del ComplexNumber.getData  
>>> c1.getData()  
Traceback (most recent call last):  
...  
AttributeError: 'ComplexNumber' object has no attribute 'getData'

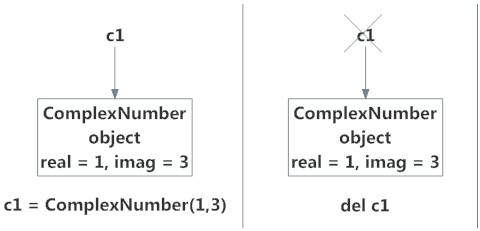
We can even delete the object itself, using the del statement.

>>> c1 = ComplexNumber(1,3)  
>>> del c1  
>>> c1  
Traceback (most recent call last):  
...  
NameError: name 'c1' is not defined

Actually, it is more complicated than that. When we do c1 = ComplexNumber(1,3), a new instance object is created in memory and the name c1 binds with it.

On the command del c1, this binding is removed and the name c1 is deleted from the corresponding namespace. The object however continues to exist in memory and if no other name is bound to it, it is later automatically destroyed.

This automatic destruction of unreferenced objects in Python is also called garbage collection.



# Python Inheritance

Inheritance enable us to define a class that takes all the functionality from parent class and allows us to add more.

## **What is Inheritance?**

Inheritance is a powerful feature in object oriented programming.

It refers to defining a new [class](https://www.programiz.com/python-programming/class) with little or no modification to an existing class. The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

### **Python Inheritance Syntax**

class BaseClass:  
 Body of base class  
class DerivedClass(BaseClass):  
 Body of derived class

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

### **Example of Inheritance in Python**

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygondefined as follows.

class Polygon:  
 def \_\_init\_\_(self, no\_of\_sides):  
 self.n = no\_of\_sides  
 self.sides = [0 for i in range(no\_of\_sides)]  
  
 def inputSides(self):  
 self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]  
  
 def dispSides(self):  
 for i in range(self.n):  
 print("Side",i+1,"is",self.sides[i])

This class has data attributes to store the number of sides, n and magnitude of each side as a list, sides.

Method inputSides() takes in magnitude of each side and similarly, dispSides() will display these properly.

A triangle is a polygon with 3 sides. So, we can created a class called Triangle which inherits from Polygon. This makes all the attributes available in class Polygon readily available in Triangle. We don't need to define them again (code re-usability). Triangle is defined as follows.

class Triangle(Polygon):  
 def \_\_init\_\_(self):  
 Polygon.\_\_init\_\_(self,3)  
  
 def findArea(self):  
 a, b, c = self.sides  
 # calculate the semi-perimeter  
 s = (a + b + c) / 2  
 area = (s\*(s-a)\*(s-b)\*(s-c)) \*\* 0.5  
 print('The area of the triangle is %0.2f' %area)

However, class Triangle has a new method findArea() to find and print the area of the triangle. Here is a sample run.

>>> t = Triangle()  
  
>>> t.inputSides()  
Enter side 1 : 3  
Enter side 2 : 5  
Enter side 3 : 4  
  
>>> t.dispSides()  
Side 1 is 3.0  
Side 2 is 5.0  
Side 3 is 4.0  
  
>>> t.findArea()  
The area of the triangle is 6.00

We can see that, even though we did not define methods like inputSides() or dispSides() for class Triangle, we were able to use them.

If an attribute is not found in the class, search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

## **Method Overriding in Python**

In the above example, notice that \_\_init\_\_() method was defined in both classes, Triangle as well Polygon. When this happens, the method in the derived class overrides that in the base class. This is to say, \_\_init\_\_() in Triangle gets preference over the same in Polygon.

Generally when overriding a base method, we tend to extend the definition rather than simply replace it. The same is being done by calling the method in base class from the one in derived class (calling Polygon.\_\_init\_\_() from \_\_init\_\_() in Triangle).

A better option would be to use the built-in function super(). So, super().\_\_init\_\_(3) is equivalent to Polygon.\_\_init\_\_(self,3) and is preferred. You can learn more about the [super() function in Python](http://rhettinger.wordpress.com/2011/05/26/super-considered-super/).

Two built-in functions isinstance() and issubclass() are used to check inheritances. Function isinstance() returns True if the object is an instance of the class or other classes derived from it. Each and every class in Python inherits from the base class object.

>>> isinstance(t,Triangle)  
True  
  
>>> isinstance(t,Polygon)  
True  
  
>>> isinstance(t,int)  
False  
  
>>> isinstance(t,object)  
True

Similarly, issubclass() is used to check for class inheritance.

>>> issubclass(Polygon,Triangle)  
False  
  
>>> issubclass(Triangle,Polygon)  
True  
  
>>> issubclass(bool,int)  
True

# **Python Multiple Inheritance**

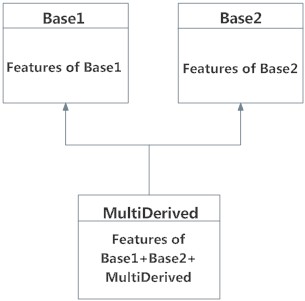
Like C++, a [class](https://www.programiz.com/python-programming/class) can be derived from more than one base classes in Python. This is called multiple inheritance.

In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single [inheritance](https://www.programiz.com/python-programming/inheritance).

### **Example**

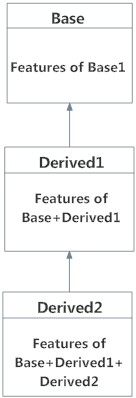
class Base1:  
 pass  
  
class Base2:  
 pass  
  
class MultiDerived(Base1, Base2):  
 pass

Here, MultiDerived is derived from classes Base1 and Base2.



The class MultiDerived inherits from both Base1 and Base2.

## **Multilevel Inheritance in Python**

On the other hand, we can also inherit form a derived class. This is called multilevel inheritance. It can be of any depth in Python.

In multilevel inheritance, features of the base class and the derived class is inherited into the new derived class.

An example with corresponding visualization is given below.

class Base:  
 pass  
  
class Derived1(Base):  
 pass  
  
class Derived2(Derived1):  
 pass

Here, Derived1 is derived from Base, and Derived2 is derived from Derived1.

## **Method Resolution Order in Python**

Every class in Python is derived from the class object. It is the most base type in Python.

So technically, all other class, either built-in or user-defines, are derived classes and all objects are instances of object class.

# Output: True

print(issubclass(list,object))

# Output: True

print(isinstance(5.5,object))

# Output: True

print(isinstance("Hello",object))

In the multiple inheritance scenario, any specified attribute is searched first in the current class. If not found, the search continues into parent classes in depth-first, left-right fashion without searching same class twice.

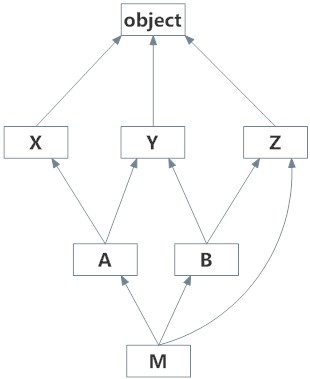
So, in the above example of MultiDerived class the search order is [MultiDerived, Base1, Base2, object]. This order is also called linearization of MultiDerived class and the set of rules used to find this order is called **Method Resolution Order (MRO)**.

MRO must prevent local precedence ordering and also provide monotonicity. It ensures that a class always appears before its parents and in case of multiple parents, the order is same as tuple of base classes.

MRO of a class can be viewed as the \_\_mro\_\_ attribute or mro() method. The former returns a tuple while latter returns a list.

>>> MultiDerived.\_\_mro\_\_  
(<class '\_\_main\_\_.MultiDerived'>,  
 <class '\_\_main\_\_.Base1'>,  
 <class '\_\_main\_\_.Base2'>,  
 <class 'object'>)  
  
>>> MultiDerived.mro()  
[<class '\_\_main\_\_.MultiDerived'>,  
 <class '\_\_main\_\_.Base1'>,  
 <class '\_\_main\_\_.Base2'>,  
 <class 'object'>]

Here is a little more complex multiple inheritance example and its visualization along with the MRO.



class X: pass

class Y: pass

class Z: pass

class A(X,Y): pass

class B(Y,Z): pass

class M(B,A,Z): pass

# Output:

# [<class '\_\_main\_\_.M'>, <class '\_\_main\_\_.B'>,

# <class '\_\_main\_\_.A'>, <class '\_\_main\_\_.X'>,

# <class '\_\_main\_\_.Y'>, <class '\_\_main\_\_.Z'>,

# <class 'object'>]

print(M.mro())

# **Python Operator Overloading**

[Python operators](https://www.programiz.com/python-programming/operators) work for built-in classes. But same operator behaves differently with different types. For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings.

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

So what happens when we use them with objects of a user-defined class? Let us consider the following class, which tries to simulate a point in 2-D coordinate system.

class Point:

def \_\_init\_\_(self, x = 0, y = 0):

self.x = x

self.y = y

Now, run the code and try to add two points in Python shell.

>>> p1 = Point(2,3)  
>>> p2 = Point(-1,2)  
>>> p1 + p2  
Traceback (most recent call last):  
...  
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'

Whoa! That's a lot of complains. TypeError was raised since Python didn't know how to add two Point objects together.

However, the good news is that we can teach this to Python through operator overloading. But first, let's get a notion about special functions.

## **Special Functions in Python**

Class functions that begins with double underscore \_\_ are called special functions in Python. This is because, well, they are not ordinary. The \_\_init\_\_() function we defined above, is one of them. It gets called every time we create a new object of that class. There are a ton of [special functions in Python](http://docs.python.org/3/reference/datamodel.html#special-method-names).

Using special functions, we can make our class compatible with built-in functions.

>>> p1 = Point(2,3)  
>>> print(p1)  
<\_\_main\_\_.Point object at 0x00000000031F8CC0>

That did not print well. But if we define \_\_str\_\_() method in our class, we can control how it gets printed. So, let's add this to our class.

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)

Now let's try the print() function again.

>>> p1 = Point(2,3)  
>>> print(p1)  
(2,3)

That's better. Turns out, that this same method is invoked when we use the built-in function str() or format().

>>> str(p1)  
'(2,3)'  
  
>>> format(p1)  
'(2,3)'

So, when you do str(p1) or format(p1), Python is internally doing p1.\_\_str\_\_(). Hence the name, special functions.

Ok, now back to operator overloading.

## **Overloading the + Operator in Python**

To overload the + sign, we will need to implement \_\_add\_\_() function in the class. With great power comes great responsibility. We can do whatever we like, inside this function. But it is sensible to return a Point object of the coordinate sum.

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)

Now let's try that addition again.

>>> p1 = Point(2,3)  
>>> p2 = Point(-1,2)  
>>> print(p1 + p2)  
(1,5)

What actually happens is that, when you do p1 + p2, Python will call p1.\_\_add\_\_(p2) which in turn is Point.\_\_add\_\_(p1,p2). Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

|  |  |  |
| --- | --- | --- |
| 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) |
| Bitwise Left Shift | p1 << p2 | p1.\_\_lshift\_\_(p2) |
| Bitwise Right Shift | p1 >> p2 | p1.\_\_rshift\_\_(p2) |
| Bitwise AND | p1 & p2 | p1.\_\_and\_\_(p2) |
| Bitwise OR | p1 | p2 | p1.\_\_or\_\_(p2) |
| Bitwise XOR | p1 ^ p2 | p1.\_\_xor\_\_(p2) |
| Bitwise NOT | ~p1 | p1.\_\_invert\_\_() |

## **Overloading Comparison Operators in Python**

Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well.

Suppose, we wanted to implement the less than symbol < symbol in our Point class.

Let us compare the magnitude of these points from the origin and return the result for this purpose. It can be implemented as follows.

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 \_\_lt\_\_(self,other):

self\_mag = (self.x \*\* 2) + (self.y \*\* 2)

other\_mag = (other.x \*\* 2) + (other.y \*\* 2)

return self\_mag < other\_mag

Try these sample runs in Python shell.

>>> Point(1,1) < Point(-2,-3)  
True  
  
>>> Point(1,1) < Point(0.5,-0.2)  
False  
  
>>> Point(1,1) < Point(1,1)  
False

Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

|  |  |  |
| --- | --- | --- |
| Operator | Expression | Internally |
| Less than | p1 < p2 | p1.\_\_lt\_\_(p2) |
| Less than or equal to | p1 <= p2 | p1.\_\_le\_\_(p2) |
| Equal to | p1 == p2 | p1.\_\_eq\_\_(p2) |
| Not equal to | p1 != p2 | p1.\_\_ne\_\_(p2) |
| Greater than | p1 > p2 | p1.\_\_gt\_\_(p2) |
| Greater than or equal to | p1 >= p2 | p1.\_\_ge\_\_(p2) |