Python 101

Object Oriented Programming In Python (Part-2)

Contents

A Recap of OOP Part-1

Class - A blueprint/ template of all objects.

Objects - An instance of a class.

- Remember the “Person” Form analogy.
- A class exists as the type “type” in python/

Objects consist of two types of attributes -

1. Data Members
2. Methods

A Recap of OOP Part-1 contd..

Data Members and Methods can be of two types-

1. Static
2. Instance

Instance attributes can be accessed using the objects only.

Static attributes can be accessed using Classes as well as the objects.

There exist classmethods as well which receive the **class** as the first argument instead of the object itself.

A Recap of OOP Part-1 contd..

isinstance - checks whether an Object is of the corresponding class or not.

is - its an operator to check equivalence of two objects. It essentially checks whether both operands have same id.

Inheritance

If we want to create some new definition (**Derived Class**) of a class which is an **extension** to other some other class (**Base Class**) then we use Inheritance.

Inhertiance is essentially a mechanism to derive a class from some existing class.

```
class DerivedClassName(BaseClassName):

   BODY_OF_DERIVED_CLASS
```

issubclass(DerivedClassName, BaseClassName) ← Returns Boolean.

Try out the following code on the interpreter

```
>>> class BaseClass:
. . . .       def method(self):
. . . .           print('Method of Base Class called')

>>> class DerivedClass(BaseClass):
. . . .       def method2(self):
. . . .           print('Method2 of Derived Class called')

>>> base = BaseClass()
>>> base.method()
Method of Base Class Called

>>> derived = DerivedClass()
>>> derived.method2()
Method2 of Derived Class called
>>> derived.method() ← Mind that it doesn't explicitly need declaring in DerivedClass as it is there in Base Class
                       and we have exntended the BaseClass to form DerivedClass
Method of Base Class called
```

Inheritance contd..

Method Overriding - Override means suspending current state and declaring an alternate. ie. redefining an existing property.

Method Overriding means redefining a method that existed in BaseClass.

Try out the following code on the interpreter

```
>>> class BaseClass:
. . . .        def method(self):
. . . .            print('Method of Base Class called')

>>> class DerivedClass(BaseClass):
. . . .        def method(self):
. . . .            print('Method of Derived Class called')

>>> base = BaseClass()
>>> base.method()
Method of Base Class Called

>>> derived = DerivedClass()
>>> derived.method()
Method of Derived Class called
```

Inheritance contd..

Multiple Inheritance - Python also supports creating classes based on multiple base classes. This type of inheritance is called Multiple Inhertiance.

MultiLevel Inheritance - Python also supports creating classes based on derived classes. This type of inheritance is called MultiLevel Inheritance.

Try out the following code on the interpreter

```
# Multiple Inheritance

>>> class BaseClass1:
. . . .       pass

>>> class BaseClass2::
. . . .        pass

>>> class DerivedClass(BaseClass1, BaseClass2):
. . . .        pass

# MultiLevel Inheritance
>>> class BaseClass:
. . . .       pass

>>> class DerivedClass1(BaseClass)::
. . . .        pass

>>> class DerivedClass2(DerivedClass1):
. . . .        pass
```

Inheritance contd..

Method Resolution Order- When we try to access some attribute of an object first python tries to find that in that object itself followed by searching for the attribute in depth-first-search manner.

The sequence in which the attributes are searched for is called **mro** or method resolution order.

We can access it using **mro()** method on objects.

Try out the following code on the interpreter

```
# Multiple Inheritance

>>> class BaseClass1:
. . . .       pass

>>> class BaseClass2::
. . . .        pass

>>> class DerivedClass(BaseClass1, BaseClass2):
. . . .        pass

>>> DerivedClass.mro()
[<class '__main__.DerivedClass’>,
<class '__main__.BaseClass1’>,
 <class '__main__.BaseClass2'>,
<class 'object'>]

# It means if an attribute is tried to be accessed it is first searched in DerivedClass, then BaseClass1 then
BaseClass2 and then object class??
```

Inheritance contd..

The Object Class- Each and every class inherits from **object class** in **Python3.**

Object class contains the __init__, __new__, __repr__, __str__ and other dunder methods.

Inheritance contd..

The Object Class- Each and every class inherits from **object class** in **Python3.**

Object class contains the __init__, __new__, __repr__, __str__ and other dunder methods.

Dunder Methods & Operator Overloading

- **Operator Overloading -** We know overloading means changing default functionality of something. Operator Overloading means changing how our Object interacts with various operators eg. +, - , /, //, * , >>, <<, <, > etc.
- **Dunder Methods -** All of the interaction of objects with Operators and some functions is based on some of the special methods in objects. These methods are called Dunder Methods. eg. __add__ method tackles how object will behave when used with + operator.

So here we can observe that in order to perform operator overloading we need to just alter body of some methods, which means in python operator overloading is achieved using Method Overloading.

Dunder Methods & Operator Overloading some important dunders

- **__init__ -** invoked when objects are initialized. Analogous to Constructors in Java.
- **__new__ -** invoked when objects are allocated memory. MUST return an object of the class, thus it is a classmethod by default.
- **__del__ -** invoked when object’s memory is freed.
- **__str__** - invoked when we try to convert objects to string. MUST return a string.
- **__repr__** - invoked when we print an object. MUST return a string.
- **__doc__ -** returns the corresponding docstring of an object.
- **__class__ -** returns the class name corresponding to an object.

Dunder Methods & Operator Overloading

dunders for Operator Overloading

- **__add__** - Overloads +
- **__sub__** - Overloads -
- **__mul__** - Overloads *
- **__lshift__** - Overloads <<
- **__rshift__** - Overloads >>
- **__pow__** - Overloads **
- **__and__** - Overloads &
- **__or__** - Overloads |
- **__xor__** - Overloads ^

Try out the following code on the interpreter

```
>>> class Point:
. . . .     def __init__(self, x, y):
. . . .         self.x = x
. . . .         self.y = y
. . . .     def __str__(self):
. . . .         return 'Point(' +str(self.x) + ' , ' +str(self.y)+ ')'
. . . .     def __add__(self, other_operand):
. . . .          if isinstance(other_operant, Point):
. . . .              return Point(self.x+other_operand.x, self.y + other_operand.y)
. . . .          else:
. . . .              return None
. . . .
>>> point1 = Point(1, 2)
>>> point2 = Point(3, 4)
>>> point1 + point2
Point( 4, 6)
```

Revisiting Exceptions user defined exceptions

When we see the MRO of any exception we find that each exception is a subclass of Exception class.

If we want to define a exception of our own and forcefully raise it inside a program we can simply create a class which extends Exception and init the class to raise it wherever needed. Like following example.

Try out the following code on the interpreter

```
>>> class CustomException(Exception):
. . . .     def __init__(self, message):
. . . .         self.message = message ← Explanation of why error occured
. . . .
>>> if True:
. . . .     raise CustomException(‘This exception occurred because of no reason at all’)
. . . .
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.CustomException: This exception occurred because of no reason at all
```

Interview Questions

How can you overload +, <<?

Using __add__ and __lshift__ method.

What kinds of Inheritances does Python support?

Single(One base one derived),

Multiple(Many base one derived),

Multilevel(sequential derived),

Heirarchical (one base many derived),

Hybrid(Mix of above).

What do you understand by MRO?

Method resolution order corresponds to the order in which base and derived classes are checked for existence of a required property.

What is the difference between __repr__ and __str__ ?

__repr__ and __str__ are quite similar methods. The only difference between both is that str's goal is to increase readability and repr's goal is to make object's unambiguous.

Thats why we most of the times see repr returning addresses etc of objects whereas str returns simple string representations that are easy to read.

__new__ is a classmethod, instance method or staticmethod?

__new__ is a classmethod. It receives the first argument as the class of the object instead of the object itself. It makes sense as well because essentially __new__ is used to create a new object by allocating memory to an object. So it would be illogical to make it an instance method when the work of __new__ is to create the object itself.