Object Oriented Programming in Python

Session -3

Contents

Difference between Procedure Oriented and Object Oriented Programming

- Procedural programming creates a step by step program that guides the application through a sequence of instructions. Each instruction is executed in order.
- Procedural programming also focuses on the idea that all algorithms are executed with functions and data that the programmer has access to and is able to change.
- Object-Oriented programming is much more similar to the way the real world works; it is analogous to the human brain. Each program is made up of many entities called objects.
- Instead, a message must be sent requesting the data, just like people must ask one another for information; we cannot see inside each other's heads.

Featuers of OOP

- Ability to simulate real-world event much more effectively
- Code is reusable thus less code may have to be written
- Data becomes active
- Better able to create GUI (graphical user interface) applications
- Programmers are able to produce faster, more accurate and better-written applications

Fundamental concepts of OOP in Python

The four major principles of object orientation are:

- Encapsulation
- Data Abstraction
- Inheritance
- Polymorphism

What is an Object..?

- Objects are the basic run-time entities in an object-oriented system.
- They may represent a person, a place, a bank account, a table of data or any item that the program must handle.
- When a program is executed the objects interact by sending messages to one another.
- Objects have two components:
  - Data (i.e., attributes)
  - Behaviors (i.e., methods)

Object Attributes and Methods Example

Object Attributes

- Store the data for that object
- Example (taxi):
  - Driver
  - OnDuty
  - NumPassengers
  - Location

Object Methods

- Define the behaviors for the object
- Example (taxi):

  - PickUp

  - DropOff

  - GoOnDuty

  - GoOffDuty

  - GetDriver

  - SetDriver

  - GetNumPassengers

What is a Class..?

- A class is a special data type which defines how to build a certain kind of object.
- The class also stores some data items that are shared by all the instances of this class
- Instances are objects that are created which follow the definition given inside of the class
- Python doesn’t use separate class interface definitions as in some languages
- You just define the class and then use it

Methods in Classes

- Define a method in a class by including function definitions within the scope of the class block
- There must be a special first argument **self** in all of method definitions which gets bound to the calling instance
- There is usually a special method called **__init__** in most classes

A Simple Class def: Student

```
class student:
""""“A class representing a student ”””””
def __init__(self , n, a):
    self.full_name = n
    self.age = a
def get_age(self):        #Method
    return self.age
```

- Define class:
  - Class name, begin with capital letter, by convention
  - object: class based on (Python built-in type)
- Define a method
  - Like defining a function
  - Must **have a special first parameter**, self, which provides way for a method to refer to object itself

Instantiating Objects with ‘__init__’

- __init__ is the default constructor
- __init__ serves as a constructor for the class. Usually does some initialization work
- An __init__ method can take any number of arguments
- However, the first argument self in the definition of __init__ is special

Self

- The first argument of every method is a reference to the current instance of the class
- By convention, we name this argument **self**
- In __init__, self refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called
- Similar to the keyword this in Java or C++
- But Python uses self more often than Java uses this
- You **do not** give a value for this parameter(self) when you call the method, Python will provide it.

Continue…

…Continue

- Although you must specify self explicitly when defining the method, you don't include it when calling the method.
- Python passes it for you automatically

Defining a method:

(this code inside a class definition.)

```
def get_age(self, num):
    self.age = num
```

Calling a method:

```
>>> x.get_age(23)
```

Deleting instances: No Need to “free”

- When you are done with an object, you don’t have to delete or free it explicitly.
- Python has automatic garbage collection.
- Python will automatically detect when all of the references to a piece of memory have gone out of scope. Automatically frees that memory.
- Generally works well, few memory leaks
- There’s also no “destructor” method for classes

Syntax for accessing attributes and methods

```
>>> f = student(“Python”, 14)

>>> f.full_name # Access attribute
“Python”

>>> f.get_age() # Access a method
14
```

Encapsulation

- Important advantage of OOP consists in the encapsulation of data. We can say that object-oriented programming relies heavily on encapsulation.
- The terms encapsulation and abstraction (also data hiding) are often used as synonyms. They are nearly synonymous, i.e. abstraction is achieved though encapsulation.
- Data hiding and encapsulation are the same concept, so it's correct to use them as synonyms
- Generally speaking encapsulation is the mechanism for restricting the access to some of an objects's components, this means, that the internal representation of an object can't be seen from outside of the objects definition.

- Access to this data is typically only achieved through special methods: ***Getters*** and ***Setters***
- By using solely get() and set() methods, we can make sure that the internal data cannot be accidentally set into an inconsistent or invalid state.
- C++, Java, and C# rely on the public, private, and protected keywords in order to implement variable scoping and encapsulation
- It's nearly always possible to circumvent this protection mechanism

Public, Protected and Private Data

- If an identifier doesn't start with an underscore character "_" it can be accessed from outside, i.e. the value can be read and changed
- Data can be protected by making members private or protected. Instance variable names starting with two underscore characters cannot be accessed from outside of the class.
- At least not directly, but they can be accessed through private name mangling.
- That means, private data __A can be accessed by the following name construct: ***instance_name._classname__A***

- If an identifier is only preceded by one underscore character, it is a protected member.
- Protected members can be accessed like public members from outside of class

Example:

```
class Encapsulation(object):
        def __init__(self, a, b, c):
                self.public = a
                self._protected = b
                self.__private = c
```

The following interactive sessions shows the behavior of public, protected and private members:

```
>>> x = Encapsulation(11,13,17)
>>> x.public
 11
>>> x._protected
13
>>> x._protected = 23
>>> x._protected
23
>>> x.__private
    Traceback (most recent call last):
              File "<stdin>", line 1, in <module>
     AttributeError: 'Encapsulation' object has no attribute '__private'
 >>>
```

The following table shows the different behavior Public, Protected and Private Data

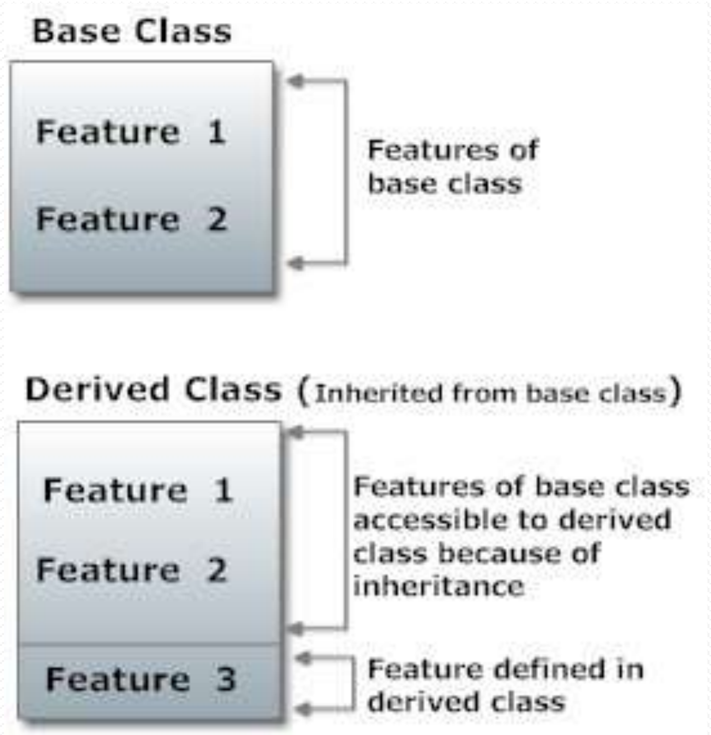
Name	Notation	Behavior
name	Public	Can be accessed from inside and outside
_name	Protected	Like a public member, but they shouldn't be directly accessed from outside
__name	Private	Can't be seen and accessed from outside

Inheritance

- Inheritance is a powerful feature in object oriented programming
- It refers to defining a new 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.
- Derived class inherits features from the base class, adding new features to it.
- This results into re-usability of code.

Syntax:

```
class Baseclass(Object):
        body_of_base_class
class DerivedClass(BaseClass):
        body_of_derived_clas
```


Base Class
Feature 1
Feature 2
Features of base class
Derived Class (Inherited from base class)
Feature 1
Feature 2
Feature 3
Features of base class accessible to derived class because of inheritance
Feature defined in derived class

While designing a inheritance concept, following key pointes keep it in mind

- A sub type never implements less functionality than the super type
- Inheritance should never be more than two levels deep
- We use inheritance when we want to avoid redundant code.

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.

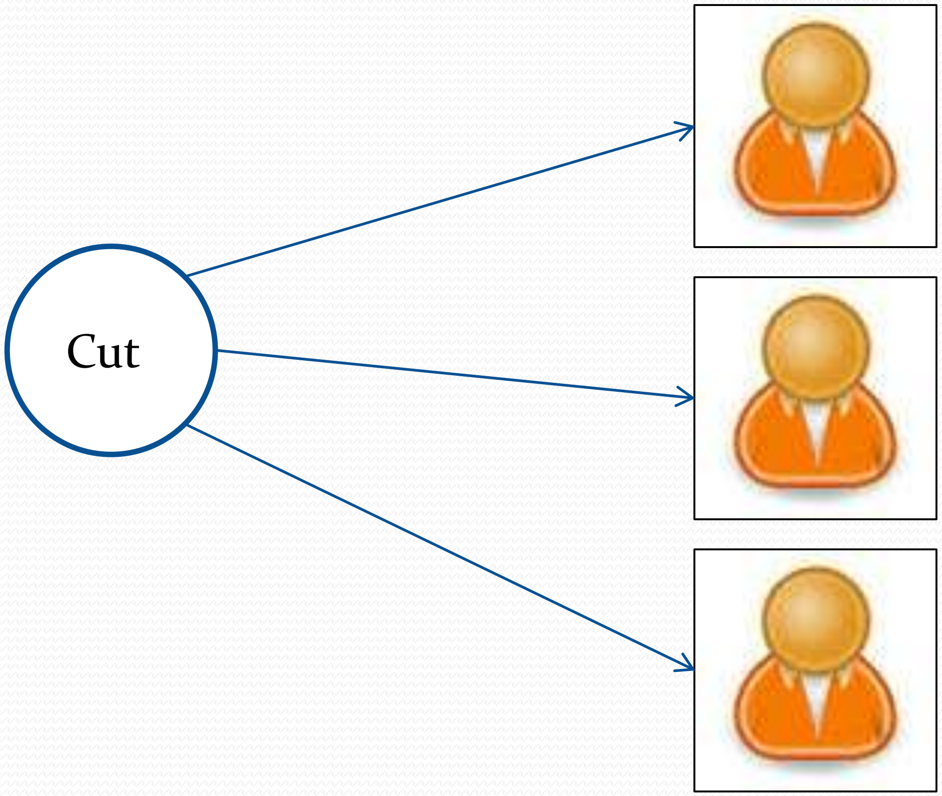
Polymorphism

- Polymorphism in Latin word which made up of ***'ploy'*** means many and ***'morphs'*** means forms
- From the Greek , Polymorphism means ***many(poly)***

  shapes (morph)
- This is something similar to a word having several different meanings depending on the context
- Generally speaking, polymorphism means that a method or function is able to cope with different types of input.

A simple word 'Cut' can have different meaning depending where it is used

If any body says "Cut" to these people



Surgeon: The Surgeon would begin to make an incision

Hair Stylist: The Hair Stylist would begin to cut someone's hair

Actor: The actor would abruptly stop acting out the current scene, awaiting directional guidance

In OOP , Polymorphism is the characteristic of being able to assign a different meaning to a particular symbol or operator in different contexts specifically to allow an entity such as a variable, a function or an object to have more than one form.

There are two kinds of Polymorphism

Overloading :

Two or more methods with different signatures

Overriding:

Replacing an inherited method with another having the same signature

Operator Overloading

- Python 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
- One final thing to mention about operator overloading is that you can make your custom methods do whatever you want. However, common practice is to follow the structure of the built-in methods.

Explanation for Operator Overloading Sample Program

What actually happens is that, when you do p1 - p2, Python will call p1.__sub__(p2) which in turn is Point.__sub__(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)