Computer Engineering Department

Advanced Python Programming(CE0620)

Classes and Object- Oriented Programming in Python

By
Prof. Bhumi Shah

Python Is Object-Oriented

- Python is a multi-paradigm programming language. It supports different programming approaches.

- One of the popular approaches to solve a programming problem is by creating objects: known as Object-Oriented Programming (OOP).
- An object has two characteristics:
  - attributes
  - behavior
  - for example, an object could represent a person with properties like a name, age, and address and behaviors such as walking, talking, breathing, and running.

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

Class in Python

- A class is a blueprint for the object.

- We can think of class as a sketch of a ”person” with labels. It contains all the details about the name, age, and address etc. Based on these descriptions, we can study about the “person”.

- The example for class of person can be :

```
class person:
    pass
```

- **class** keyword is used to define an empty class person.
- 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 “person” class can be:

```
obj1 = person()
```

Here, obj1 is an object of class person.

Example

```
class person:
  age = 50

p1 = person()
print(p1.age)
```

__init__() Function

- built-in __init__() function
- the method the __init__() simulates the constructor of the class
- All classes have a function called __init__(), which is always executed when the class is being initiated.
- The properties that all person objects must have are defined in a method called .__init__().
- Every time a new person object is created, .__init__() sets the initial state of the object by assigning the values of the object’s properties.
- It accepts the ***self***-keyword as a first argument which allows accessing the attributes or method of the class.
- When a new class instance is created, the instance is automatically passed to the self parameter in .__init__() so that new attributes can be defined on the object.

Example

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("ABC", 50)
“””To instantiate objects of this person class,
you need to provide values for the name and
age. If you don’t, then Python raises a
TypeError:”””
print(p1.name)
print(p1.age)
```

The __init__() function is called automatically every time the class is being used to create a new object.

- self.name = name creates an attribute called name and assigns to it the value of the name parameter.

- self.age = age creates an attribute called age and assigns to it the value of the age parameter.

Example

```
class Employee:
    def __init__(self, name, id):
        self.id = id
        self.name = name

    def display(self):
        print("ID: %d \nName: %s" % (self.id, self.name))


emp1 = Employee("XYZ”, 101)
emp2 = Employee("ABC", 102)

emp1.display()

emp2.display()
```

__init__() Function

- Attributes created in .__init__() are called instance attributes.
- An instance attribute’s value is specific to a particular instance of the class. All person objects have a name and an age, but the values for the name and age attributes will vary depending on the person instance.

- On the other hand, class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of .__init__().

Instantiate an Object in Python

- Creating a new object from a class is called instantiating an object.
- We can instantiate a new **person** object by typing the name of the class, followed by opening and closing parentheses:

  - person()
  - p1=person()

Instance Methods

- Instance methods are functions that are defined inside a class and can only be called from an instance of that class.
- like .__init__(), an instance method’s first parameter is always self.

```
class animal:
    species = "Canis"

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

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

.__str__() method

- When we print(p1), it displays message telling you that ***p1 is a person object at the memory address 0x00aeff70***.
- This message can be changed what gets printed by defining a special instance method called .__str__().

```
class person:
    # Leave other parts of class as-is


    # Replace .description() with __str__()
    def __str__(self):
        return f"{self.name} is {self.age} years old"
```

Note: *.__init__() and .__str__() are called dunder methods because they begin and end with double underscores*.

Abstract Data Types and Classes

- The abstract data type is special kind of data type, whose behavior is defined by a set of values and set of operations.
- The keyword “Abstract” is used as we can use these data types, we can perform different operations
- But how those operations are working that is totally hidden from the user.
- The ADT is made of primitive data types, but operation logics are hidden.

Inheritance

- The method of inheriting the properties of parent class into a child class is known as inheritance. It is an OOP concept.
- benefits of inheritance.

  - Code reusability- we do not have to write the same code again and again, we can just inherit the properties we need in a child class.

  - It represents a real world relationship between parent class and child class.

  - It is transitive in nature. If a child class inherits properties from a parent class, then all other sub-classes of the child class will also inherit the properties of the parent class.

Steps To perform inheritance

1. Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class

```
class Parent():
```

2. Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class

```
class Child(Parent):
```

Example

```
class Parent():
    def first(self):
        print('Parent’s function')

class Child(Parent):
    def second(self):
        print('Child’s function')

ob = Child()
ob.first()
ob.second()
```

__init__() in inheritance

- The __init__() function is called every time a class is being used to make an object.

- When we add the __init__() function, the child class will no longer inherit the parent's __init__() function.

- The child’s class __init__() function overrides the parent class’s __init__() function.

- To keep the inheritance of the parent's __init__() function, we need to add a call to the parent's __init__() function

```
class Parent:
    def __init__(self , fname, fage):
        self.firstname = fname
        self.age = fage
    def view(self):
        print(self.firstname , self.age)

class Child(Parent):
    def __init__(self , fname , fage):
        Parent.__init__(self, fname, fage)
        self.lastname = "ChildClass"
    def view(self):
        print("child name" , self.firstname ,"has the ",  self.age , "
age." , self.lastname, ":Testing")
ob = Child("XYZ" , '32')
ob.view()
```

Python - Public, Protected, Private Members

- **Public Members**:accessible from outside the class.
- The object of the same class is required to invoke a public method.
- This arrangement of private instance variables and public methods ensures the principle of data encapsulation.
- All members in a Python class are public by default.

Example

```
class Student:
    schoolName = 'XYZ School' # class attribute

    def __init__(self, name, age):
        self.name=name # instance attribute
        self.age=age # instance attribute

std = Student("ABC", 25)
std.schoolName

std.name

std.age = 20
std.age
```

Python - Public, Protected, Private Members

- **Protected Members:** Protected members of a class are accessible from within the class and are also available to its sub-classes.
- No other environment is permitted access to it.
- This enables specific resources of the parent class to be inherited by the child class.
- Python's convention to make an instance variable protected is to add a prefix _ (single underscore) to it.
- This effectively prevents it from being accessed unless it is from within a sub-class.

Example

```
class Student:
    _schoolName = 'XYZ School' # protected class attribute

    def __init__(self, name, age):
        self._name=name  # protected instance attribute
        self._age=age # protected instance attribute

 std = Student("Swati", 25)
std._name

std._name = 'Dipa'
std._name
```

Python - Public, Protected, Private Members

- **Private Members**: Python doesn't have any mechanism that effectively restricts access to any instance variable or method.
- Python prescribes a convention of prefixing the name of the variable/method with a single or double underscore to emulate the behavior of protected and private access specifiers.

- The double underscore __ prefixed to a variable makes it private.
- It gives a strong suggestion not to touch it from outside the class.
- Any attempt to do so will result in an AttributeError:

Example

```
class Student:
    __schoolName = 'XYZ School' # private class attribute

    def __init__(self, name, age):
        self.__name=name  # private instance attribute
        self.__age=age # private instance attribute
    def __display(self):  # private method
                print('This is private method.')
std = Student("Bill", 25)
std.__schoolName
AttributeError: 'Student' object has no attribute '__schoolName'
std.__name
AttributeError: 'Student' object has no attribute '__name'
std.__display()
AttributeError: 'Student' object has no attribute '__display'
```

super() Function

- The super() builtin method used to call the super claa constructor or methods from the sub class.
- Allows us to avoid using the base class name explicitly
- Working with Multiple Inheritance

Syntax:

```
        super().__init__()
        super().__init__(arguments)
```

we can also call super class methods

```
        super().function1()
```

Example

```
class A(object):
  def __init__(self, AName):
    print(AName, ' is Super Class.')

class B(A):
  def __init__(self):
    print('This is Child Class')
    super().__init__('A')

ob=B()
```

“Object” represents the base class name from where all classes in Python are derived.Its not compulsory to write it.

Types of Inheritance in Python

There are two types of Inheritance:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- hierarchical inheritance

Single Inheritance

- When a child class inherits only a single parent class.

```
class Parent:
    def func1(self):
        print("this is function one")
class Child(Parent):
    def func2(self):
        print(" this is function 2 ")
ob = Child()
ob.func1()
ob.func2()
```

Multiple Inheritance

- When a child class inherits from more than one parent class.

```
class Parent:
   def func1(self):
        print("this is function 1")
class Parent2:
   def func2(self):
        print("this is function 2")
class Child(Parent , Parent2):
    def func3(self):
        print("this is function 3")

ob = Child()
ob.func1()
ob.func2()
ob.func3()
```

Problems in Multiple inheritance

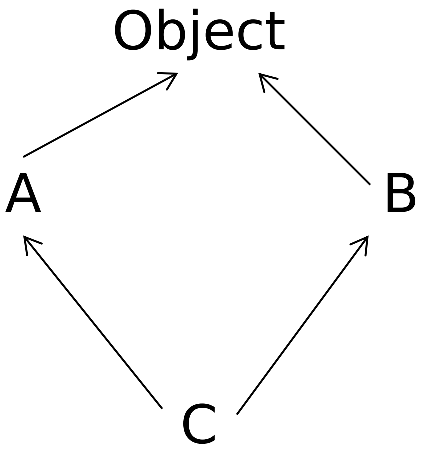
```
class Class1:
    def m(self):
        print("In Class1")

class Class2(Class1):
    def m(self):
        print("In Class2")

class Class3(Class1):
    def m(self):
        print("In Class3")

class Class4(Class2, Class3):
    pass
obj = Class4()
obj.m()
```

Problems in Multiple inheritance

```
class A(object):
    def __init__(self):
        self.a="a"
        print(self.a)

class B(object):
    def __init__(self):
        self.b="b"
        print(self.b)

class C(A,B):
    def __init__(self):
        self.c="c"
        print(self.c)
        super().__init__()

ob=C()
```

Solution

```
class A(object):
    def __init__(self):
        self.a="a"
        print(self.a)
        super().__init__()
class B(object):
    def __init__(self):
        self.b="b"
        print(self.b)
        super().__init__()
class C(A,B):
    def __init__(self):
        self.c="c"
        print(self.c)
        super().__init__()
ob=C()
```



MRO-Method Resolution Order

- A method is serched first in current class.
- if not there,it will continue the search in parents claas from left to right fashion,in depth-first search.

1. search into the child class/sub class before going for the parent class.
2. in base classes ,it search from left to right fashion,in depth-first search.
3. It will not visit any class more than once.

Multilevel Inheritance

- When a child class becomes a parent class for another child class.

```
class Parent:
    def func1(self):
        print("this is function 1")
class Child(Parent):
    def func2(self):
        print("this is function 2")
class Child2(Child):
    def func3("this is function 3")
ob = Child2()
ob.func1()
ob.func2()
ob.func3()
```