3. Python Object Oriented Programming

3.1. Concept of class, object and instances, method call

3.2. Constructor, class attributes and destructors

3.3. Real time use of class in live projects

3.4. Inheritance, super class and overloading operators,

3.5. Static and class methods

3.6. Adding and retrieving dynamic attributes of classes

3.7. Programming using OOPS

3.8. Delegation and container

3.1. Concept of class, object and instances, method call

3.2. Constructor, class attributes and destructors

3.3. Real time use of class in live projects

3.4. Inheritance, super class and overloading operators

Python Class

Python is a completely object-oriented language. You have been working with classes and objects right from the beginning of these tutorials. Every element in a Python program is an object of a class. A number, string, list, dictionary, etc., used in a program is an object of a corresponding built-in class. You can retrieve the class name of variables or objects using the type() method, as shown below.

Example: Python Built-in Classes

```
>>> num=20
>>> type(num)
<class 'int'>
>>> s="Python"
>>> type(s)
<class 'str'>
```

Defining a Class

A class in Python can be defined using the class keyword.

```
class <ClassName>:
    <statement1>
    <statement2>
    .
    .
    <statementN>
```

As per the syntax above, a class is defined using the **class** keyword followed by the class name and **:** operator after the class name, which allows you to continue in the next indented line to define class members. The followings are **class members.**

1. Class Attributes
2. Constructor
3. Instance Attributes
4. Properties
5. Class Methods

A class can also be defined without any members. The following example defines an empty class using the pass keyword.

Example: Define Python Class

```
class Student:
    pass
```

Class instantiation uses function notation. To create an object of the class, just call a class like a parameter-less function that returns a new object of the class, as shown below.

Example: Creating an Object of a Class

```
std = Student()
```

Above, Student() returns an object of the Student class, which is assigned to a local variable std. The Student class is an empty class because it does not contain any members.

Class Attributes

Class attributes are the variables defined directly in the class that are shared by all objects of the class. Class attributes can be accessed using the class name as well as using the objects.

Example: Define Python Class

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

Above, **the schoolName is a class attribute defined inside a class. The value of the schoolName will remain the same for all the objects unless modified explicitly.**

Example: Define Python Class

```
>>> Student.schoolName
```

```
'XYZ School'
>>> std = Student()
>>> std.schoolName
'XYZ School'
```

As you can see, a class attribute is accessed by `Student.schoolName` as well as `std.schoolName`. **Changing the value of class attribute using the class name would change it across all instances. However, changing class attribute value using instance will not reflect to other instances or class.**

Example: Define Python Class

```
>>> Student.schoolName = 'ABC School' # change attribute value using class name
>>> std = Student()
>>> std.schoolName
'ABC School'   # value changed for all instances
>>> std.schoolName = 'My School'  # changing instance's attribute
>>> std.schoolName
'My School'
>>> Student.schoolName # instance level change not reflected to class attribute
'ABC School'
>>> std2 = Student()
>>> std2.schoolName
'ABC School'
```

The following example demonstrates the use of class attribute `count`.

Example: Student.py

```
class Student:
    count = 0
    def __init__(self):
        Student.count += 1
```

In the above example, **`count` is an attribute in the Student class**. Whenever a new object is created, the value of **`count`** is incremented by 1. You can now access **the `count`** attribute after creating the objects, as shown below.

Example:

```
>>> std1=Student()
>>> Student.count
1
>>> std2 = Student()
>>> Student.count
2
```

Constructor

In Python, the constructor method is invoked automatically whenever a new object of a class is instantiated, same as constructors in C# or Java. The constructor must have a special name `__init__()` and a special parameter called `self`.

Note:

The first parameter of each method in a class must be the `self`, which refers to the calling object. However, you can give any name to the first parameter, not necessarily `self`.

The following example defines a constructor.

Example: Constructor

```
class Student:
    def __init__(self): # constructor method
        print('Constructor invoked')
```

Now, whenever you create an object of the `Student` class, the `__init__()` constructor method will be called, as shown below.

Example: Constructor Call on Creating Object

```
>>>s1 = Student()
Constructor invoked
>>>s2 = Student()
Constructor invoked
```

The constructor in Python is used to define the attributes of an instance and assign values to them.

Destructor

Destructor is also a special method gets executed automatically when an object exit from the scope. It is just opposite to constructor. In Python, __del__( ) method is used as destructor.

Example : Program to illustrate about the __del__( ) method

```
# Example - Program to illustrate about the __del__( ) method

class Sample:
    count = 0

    def __init__(self, var):
        Sample.count += 1
        self.var = var
        print("The object value is = ", var)
        print("The value of class variable is = ", Sample.count)

    def __del__(self):
        Sample.count -= 1
        print(f'Object with value {self.var} is exit from the scope')
```

Input:

```
s1 = Sample(50)
s2 = Sample(15)
s3 = Sample(40)
```

Output:

```
The object value is = 50
The value of class variable is = 1
Object with value 50 is exit from the scope
The object value is = 15
The value of class variable is = 1
Object with value 15 is exit from the scope
```

The object value is = 40

The value of class variable is = 1

Object with value 40 is exit from the scope

Instance Attributes

Instance attributes are attributes or properties attached to an instance of a class. Instance attributes are defined in the constructor.

The following example defines instance attributes name and age in the constructor.

Example: Instance Attributes

```
class Student:
    schoolName = 'XYZ School' # class attribute
    def __init__(self): # constructor
        self.name = " Milind" # instance attribute
        self.age = 40 # instance attribute
```

An instance attribute can be accessed using **dot notation: [instance name].[attribute name]**, as shown below.

Example:

```
>>> std = Student()
>>> std.name
Milind
>>> std.age
40
```

You can set the value of attributes using the dot notation, as shown below.

Example:

```
>>> std = Student()
>>> std.name = "Bill" # assign value to instance attribute
>>> std.age=25       # assign value to instance attribute
>>> std.name         # access instance attribute value
Bill
>>> std.age          # access value to instance attribute
25
```

You can specify the values of **instance attributes** through **the constructor**. The following constructor includes the name and age parameters, other than the self parameter.

Example: Setting Attribute Values

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

Now, you can specify the values while creating an instance, as shown below.

Example: Passing Instance Attribute Values in Constructor

```
>>> std = Student('Bill',25)
>>> std.name
'Bill'
>>> std.age
25
```

Note:

You don't have to specify the value of the self parameter. It will be assigned internally in Python.

You can also set default values to the instance attributes. The following code sets the default values of the constructor parameters. So, if the values are not provided when creating an object, the values will be assigned latter.

Example: Setting Default Values of Attributes

```
class Student:
    def __init__(self, name="Guest", age=25)
        self.name=name
        self.age=age
```

Now, you can create an object with default values, as shown below.

Example: Instance Attribute Default Value

```
>>> std = Student()
>>> std.name
'Guest'
>>> std.age
```

25

Class Properties

In Python, a property in the class can be defined using the property() function.

The property() method in Python provides **an interface to instance attributes**. It encapsulates instance attributes and provides a property, same as Java and C#.

The property() method takes the get, set and delete methods as arguments and returns an object of the property class.

The following example demonstrates how to create a property in Python using the property() function.

Example: property()

```
class Student:
    def __init__(self):
        self.__name=''
    def setname(self, name):
        print('setname() called')
        self.__name=name
    def getname(self):
        print('getname() called')
        return self.__name
    name=property(getname, setname)
```

In the above example, property(getname, setname) returns the property object and assigns it to name. Thus, the name property hides the private instance attribute __name. The name property is accessed directly, but internally it will invoke the getname() or setname() method, as shown below.

Example: property()

```
>>> std = Student()
>>> std.name="Steve"
setname() called
>>> std.name
getname() called
```

'Steve'

It is recommended to use the **property decorator** instead of the property() method.

Python Property Decorator - @property

The @property decorator is a built-in decorator in Python for the **property() function**. Use @property decorator on any method in the class to use the method as a property.

You can use the following three decorators to define a property:

- @property: Declares the method as a property.
- @<property-name>.setter: Specifies the setter method for a property that sets the value to a property.
- @<property-name>.deleter: Specifies the delete method as a property that deletes a property.

Declare a Property

The following declares the method as a property. This method must return the value of the property.

Example: @property decorator

```
class Student:
    def __init__(self, name):
        self.__name = name
    @property
    def name(self):
        return self.__name
```

Above, @property decorator applied to the name() method. The name() method returns the **private** instance attribute value **__name**. So, we can now use the name() method as a property to get the value of the **__name** attribute, as shown below.

Example: Access Property decorator

```
>>> s = Student('Steve')
>>> s.name
'Steve'
```

Property Setter

Above, we defined the name() method as a property. We can only access the value of the name property but cannot modify it. To modify the property value, we must define the setter method for the name property using @property-name.setter decorator, as shown below.

Example: Property Setter

```
class Student:
    def __init__(self, name):
        self.__name = name

    @property
    def name(self):
        return self.__name
    @name.setter   #property-name.setter decorator
    def name(self, value):
        self.__name = value
```

Above, we have two overloads of the name() method. One is for the getter and another is the setter method. The setter method must have the value argument that can be used to assign to the underlying private attribute. Now, we can retrieve and modify the property value, as shown below.

Example: Access Property

```
>>> s = Student('Steve')
>>> s.name
'Steve'
>>> s.name = 'Bill'
'Bill'
```

Property Deleter

Use the @property-name.deleter decorator to define the method that deletes a property, as shown below.

Example: Property Deleter

```
class Student:
    def __init__(self, name):
        self.__name = name
    @property
    def name(self):
        return self.__name
    @name.setter
    def name(self, value):
        self.__name = value
    @name.deleter   #property-name.deleter decorator
    def name(self):
        print('Deleting..')
        del self.__name
```

The **deleter** would be invoked when you delete the property using keyword **del,** as shown below. Once you delete a property, you cannot access it again using the same instance.

Example: Delete a Property

```
>>> s = Student('Steve')
>>> del s.name
Deleting..
>>> s.name
Traceback (most recent call last):
File "<pyshell#16>", line 1, in <module>
    p.name
File "C:\Python37\test.py", line 6, in name
    return self.__name
AttributeError: 'Student' object has no attribute '_Student__name'
```

Class Methods

You can define **as many methods as** you want in a class using the **def** keyword. Each method must have the first parameter, generally named as **self,** which refers to the calling instance.

Example: Class Method

```
class Student:
    def displayInfo(self): # class method
        print('Student Information')
```

Self is just a conventional name for the first argument of a method in the class. A method defined as **mymethod(self, a, b)** should be called as **x.mymethod(a, b)** for the object **x** of the class.

The above class method can be called as a normal function, as shown below.

Example: Class Method

```
>>> std = Student()
>>> std.displayInfo()
'Student Information'
```

The first parameter of the method need not be named **self.** You can give any name that refers to the instance of the calling method. The following displayInfo() method names the first parameter as **obj** instead of **self** and that works perfectly fine.

Example: Class Method

```
class Student:
    def displayInfo(obj): # class method
        print('Student Information')
```

Defining a method in the class without the self parameter would raise an exception when calling a method.

Example: Class Method

```
class Student:
    def displayInfo(): # method without self parameter
        print('Student Information')
>>> std = Student()
>>> std.displayInfo()
```

```
Traceback (most recent call last):
std.displayInfo()
TypeError: displayInfo() takes 0 positional arguments but 1 was given
```

The method can access instance attributes using the self parameter.

Example: Class Method

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def displayInfo(self): # class method
        print('Student Name: ', self.name,', Age: ', self.age)
```

You can now invoke the method, as shown below.

Example: Calling a Method

```
>>> std = Student('Steve', 25)
>>> std.displayInfo()
Student Name: Steve , Age: 25
```

Deleting Attribute, Object, Class

You can delete attributes, objects, or the class itself, using the del keyword, as shown below.

Example: Delete Attribute, Object, Class

```
>>> std = Student('Steve', 25)
>>> del std.name # deleting attribute
>>> std.name
Traceback (most recent call last):
File "<pyshell#42>", line 1, in <module>
std.name
AttributeError: 'Student' object has no attribute 'name'
>>> del std  # deleting object
>>> std.name
Traceback (most recent call last):
File "<pyshell#42>", line 1, in <module>
```

```
std.name
NameError: name 'std' is not defined
>>> del Student  # deleting class
>>> std = Student('Steve', 25)
Traceback (most recent call last):
File "<pyshell#42>", line 1, in <module>
std = Student()
NameError: name 'Student' is not defined
```

Inheritance in Python

We often come across different products that have **a basic model** and **an advanced model** with added features over and above basic model. **A software modelling approach of OOP enables extending the capability of an existing class to build a new class, instead of building from scratch.** In OOP terminology, this characteristic is called as **inheritance**; the existing class is called **base or parent class**, while the **new class** is called **child or sub class.**

Inheritance comes into picture when **a new class possesses** the **'IS A'** relationship with an existing class.

Dog **IS** an animal. Cat also **IS** an animal. Hence, **animal** is the **base class**, while **dog** and **cat** are **inherited classes.**

A quadrilateral has four sides. A **rectangle** IS **a quadrilateral**, and also **square** IS **a quadrilateral**. **Quadrilateral** is **a base class** (also called **parent class**), while **rectangle** and **square** are the **inherited classes** - also called **child classes.**

The **child class** inherits **data definitions** and **methods** from the **parent class**. This facilitates the **reuse of features already available**. The **child class** can add **a few more definitions** or **redefine a base class method.**

This feature is extremely useful in building a hierarchy of classes for objects in a system. It is also possible to design **a new class** based upon **more than one existing classes**. This feature is called **Multiple inheritances.**

The general mechanism of establishing inheritance is illustrated below:

Syntax:

```
class parent:
    statements

class child(parent):
    statements
```

While defining the child class, **the name of the parent class is put in the parentheses** in front of it, indicating the relation between the two. Instance attributes and methods defined in the parent class will be inherited by **the object** of the **child class.**

To demonstrate a more meaningful example, a quadrilateral class is first defined, and it is used as **a base class** for **the rectangle class.**

A quadrilateral class having four sides as instance variables and **a perimeter()** method is defined below:

Example:

```
class quadriLateral:
    def __init__(self, a, b, c, d):
        self.side1=a
        self.side2=b
        self.side3=c
        self.side4=d
    def perimeter(self):
        p=self.side1 + self.side2 + self.side3 + self.side4
        print("perimeter=", p)
```

The constructor (the `__init__()` method) receives four parameters and assigns them to four instance variables. To test the above class, declare its object and invoke the `perimeter()` method.

```
>>>q1=quadriLateral(7,5,6,4)
```

```
>>>q1.perimeter()
```

```
perimeter=22
```

We now design a rectangle class based upon the quadriLateral class (rectangle IS a quadrilateral!). The **instance variables** and **the perimeter()** method from the base class should be automatically available to it without redefining it.

Since opposite sides of the rectangle are the same, we need **only two adjacent** sides to construct its object. Hence, the other two parameters of the __init__() method **are set to none.** The **__init__()** method forwards the parameters to the constructor of its base (quadrilateral) class using the **super()** function. The object is initialized with side3 and side4 set to none. Opposite sides are made equal by the constructor of rectangle class. Remember that it has automatically inherited the perimeter() method, hence there is no need to redefine it.

Example: Inheritance

```
class rectangle(quadriLateral):
    def __init__(self, a, b):
        super().__init__(a, b, a, b)
```

We can now declare the object of the rectangle class and call the perimeter() method.

```
>>> r1=rectangle(10, 20)
```

```
>>> r1.perimeter()
```

```
perimeter=60
```

Overriding in Python

In the above example, we see how resources of the base class are reused while constructing the inherited class. However, the inherited class can have its own instance attributes and methods.

Methods of the parent class are available for use in the inherited class. However, if needed, we can modify the functionality of any base class method. For that purpose, the inherited class contains a new definition of a method (with the same name and the signature already present in the base class). Naturally, the object of a new class will have access to both methods, but the one from its own class will have precedence when invoked. This is called method overriding.

First, we shall define a new method named area() in the rectangle class and use it as a base for the square class. The area of rectangle is the product of its adjacent sides.

Example:

```
class rectangle(QuadriLateral):
    def __init__(self, a,b):
        super().__init__(a, b, a, b)

    def area(self):
        a = self.side1 * self.side2
        print("area of rectangle=", a)
```

Let us define the square class which inherits the rectangle class. The area() method is overridden to implement the formula for the area of the square as the square of its sides.

Example:

```
class square(rectangle):
    def __init__(self, a):
        super().__init__(a, a)
    def area(self):
        a=pow(self.side1, 2)
        print('Area of Square: ', a)
```

```
>>>s=Square(10)
```

```
>>>s.area()
```

```
Area of Square: 100
```

Operator overloading in Python

Operators are used in Python to perform specific operations on the given operands. The operation that any particular operator will perform on any predefined data type is already defined in Python.

Each operator can be used in a different way for different types of operands. For example, + operator is used for **adding two integers** to give an integer as a result but when we use it with **float operands**, then the result is a float value and when + is used with **string operands** then it concatenates the two operands provided.

This different behavior of a single operator for different types of operands is called **Operator Overloading**. The use of + operator with different types of operands is shown below:

```
>>> x = 10
>>> y = 20
>>> x + y
30
>>> z = 10.4
>>> x + z
20.4
>>> s1 = 'hello'
>>> s2 = 'world'
>>> s1 + s2
'hello world'
```

Can + Operator Add anything?

The answer is No, it cannot. Can you use the + operator to add two objects of a class. The + operator can add two integer values, two float values or can be used to concatenate two strings only because these behaviors have been defined in python.

So if you want to use the same operator to add two objects of some user defined class then you will have to defined that behavior yourself and inform python about that.

If you are still not clear, let's create a class and try to use the + operator to add two objects of that class,

```
class Complx:

    def __init__(self, r, i):

        self.real = r

        self.img = i

c1 = Complx(5, 3)

c2 = Complx(2, 4)

print("sum = ", c1 + c2)
```

Output:

```
TypeError Traceback (most recent call last)
<ipython-input-7-4898e03f9a7e> in <module>
      5 c1 = Complx(5, 3)
      6 c2 = Complx(2, 4)
----> 7 print("sum = ", c1 + c2)
TypeError: unsupported operand type(s) for +: 'Complx' and 'Complx'
```

So we can see that the + operator is not supported in a user - defined class. But we can do the same by overloading the + operator for our class Complx. But how can we do that?

Special Functions in Python

Special functions in python are the functions which are used to perform special tasks. These special functions have __ as prefix and suffix to their name as we see in __init__() method which is also a special function. Some special functions used for **overloading the operators are shown below:**

Mathematical Operator

Below we have the names of the special functions to overload the mathematical operators in python.

Name	Symbol	Special Function
Addition	+	__add__(self, other)
Subtraction	-	__sub__(self, other)
Division	/	__truediv__(self, other)
Floor Division	//	__floordiv__(self, other)
Modulus(or Remainder)	%	__mod__(self, other)
Power	**	__pow__(self, other)

Assignment Operator

Below we have the names of the special functions to overload the assignment operators in python.

Name	Symbol	Special Function
Increment	**+=**	**__iadd__(self, other)**
Decrement	**-=**	**__isub__(self, other)**
Product	***=**	**__imul__(self, other)**
Division	**/=**	**__idiv__(self, other)**
Modulus	**%=**	**__imod__(self, other)**
Power	****=**	**__ipow__(self, other)**

Relational Operator

Below we have the names of the special functions to overload the relational operators in python.

Name	Symbol	Special Function
Less than	**<**	**__lt__(self, other)**
Greater than	**>**	**__gt__(self, other)**
Equal to	**==**	**__eq__(self, other)**
Not equal	**!=**	**__ne__(self, other)**
Less than or equal to	**<=**	**__le__(self, other)**
Greater than or equal to	**> =**	**__ge__(self, other)**

It's time to see a few code examples where we actually use the above specified special functions and overload some operators.

Overloading + operator

In the below code example we will overload the + operator for our class Complex,

```
class Complx:
    # defining init method for class
    def __init__(self, r, i):
        self.real = r
        self.img = i


    # string function to print object of Complx class
```

```
    def __str__(self):
        return f"{self.real} + {self.img}j"

    # overloading the add operator using special function
    def __add__(self, other):
        r = self.real + other.real
        i = self.img + other.img
        return Complx(r, i)

c1 = Complx(5, 8)
c2 = Complx(2, 5)
print("sum = ", c1 + c2)
```

Output:

```
sum =  7 + 13j
```

In the program above, __add__() is used to overload the + operator i.e. when + operator is used with two Complx class objects then the function __add__() is called.

__str__() is another special function which is used to provide a format of the object that is suitable for printing.

Overloading < operator

Now let's overload the less than operator so that we can easily compare two Complex class object's values by using the less than operation <.

As we know now, for doing so, we have to define the __lt__ special function in our class.

```
class Complex:
    # defining init method for class
    def __init__(self, r, i):
        self.real = r
        self.img = i
```

```
    # overloading the less than operator using special function
    def __lt__(self, other):
        if self.real < other.real:
            return true
        elif self.real == other.real:
            if self.img < other.real:
                return True
            else:
                return False
        else:
            return False


c1 = Complex(5,4)
c2 = Complex(4,3)
print("Is c1 less than c2: ",c1 < c2)
```

Output:

```
Is c1 less than c2:  False
```

Python - Magic or Dunder Methods

Magic methods in Python are the **special methods** that start and end with the **double underscores.** They are also called dunder methods. Magic methods are not meant to be invoked directly by you, but the invocation happens internally from the class on a certain action. For example, when you add two numbers using the + operator, internally, the __add__() method will be called.

Built-in classes in Python define many magic methods. Use the dir() function to see the number of magic methods inherited by a class. For example, the following lists all the attributes and methods defined in the int class.

Python - Public, Protected, Private Members

Classical object-oriented languages, such as C++ and Java, control the access to class resources by **public, private**, and **protected** keywords. Private members of the class are denied access from the environment outside the class. They can be handled only from within the class.

Public Members

Public members (generally methods declared in a class) are 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. Any member can be accessed from outside the class environment.

Example: Public Attributes

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

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

You can access the Student class's attributes and also modify their values, as shown below.

Example: Access Public Members

```
>>> std = Student("Steve", 25)
>>> std.schoolName
'XYZ School'
>>> std.name
'Steve'
>>> std.age = 20
>>> std.age
20
```

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: Protected Attributes

```
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
```

In fact, this doesn't prevent instance variables from accessing or modifying the instance. You can still perform the following operations:

Example: Access Protected Members

```
>>> std = Student("Swati", 25)
>>> std._name
'Swati'
>>> std._name = 'Dipa'
>>> std._name
'Dipa'
```

However, you can define a property using property decorator and make it protected, as shown below.

Example: Protected Attributes

```
class Student:
        def __init__(self,name):
                self._name = name
        @property
```

```
    def name(self):
        return self._name
    @name.setter
    def name(self, newname):
        self._name = newname
```

Above, @property decorator is used to make the `name()` method as property and `@name.setter` decorator to another overloads of the `name()` method as property setter method. Now, `_name` is protected.

Example: Access Protected Members

```
>>> std = Student("Swati")
>>> std.name
'Swati'
>>> std.name = 'Dipa'
>>> std.name
'Dipa'
>>> std._name # still accessible
```

Above, we used `std.name` property to modify `_name` attribute. However, it is still accessible in Python. Hence, the responsible programmer would refrain from accessing and modifying instance variables prefixed with `_` from outside its class.

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: Private Attributes

```
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.')
```

Example:

```
>>> 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'
```

Python performs **name mangling** of **private variables**. Every member with a double underscore will be changed to `_object._class__variable`. So, it can still be accessed from outside the class, but the practice should be refrained.

Example:

```
>>> std = Student("Bill", 25)
>>> std._Student__name
'Bill'
>>> std._Student__name = 'Steve'
>>> std._Student__name
'Steve'
>>> std._Student__display()
'This is private method.'
```

Thus, Python provides conceptual implementation of public, protected, and private access modifiers, but not like other languages like C#, Java, C++.

 # Example - Program to illustrate public, protected and private variables

```
class Sample:
```

```
    def __init__(self, n1, n2, n3):
        self.n1 = n1      # public instance attribute
        self._n2 = n2     # protected instance attribute
        self.__n3 = n3  # private instance attribute

    def display(self):
        print("Class - Public variable = ", self.n1)
        print("Class - Protected variable = ", self._n2)
        print("Class - Private variable = ", self.__n3)

s = Sample(10, 15, 25)
```

s.display()

```
print("Public variable = ", s.n1)
print("Protected variable = ", s._n2)
print("Private variable = ", s.__n3)
```

Output:

```
Class - Public variable =  10
Class - Protected variable =  15
Class - Private variable =  25
Public variable =  10
Protected variable =  15
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-45-2046ea42bcfe> in <module>
     19 print("Public variable = ", s.n1)
     20 print("Protected variable = ", s._n2)
---> 21 print("Private variable = ", s.__n3)

AttributeError: 'Sample' object has no attribute '__n3'
```

 # Example: Sample Programs to illustrate classes and objects

 # Write a program to calculate area and circumference of a circle

```
class Circle:
    pi=3.14
   def __init__(self,radius):
```

```
        self.radius = radius
    def area(self):
        a = Circle.pi * (self.radius**2)
        print(f'The area of Circle = {a}')
    def circumference(self):
        c = round((2 * Circle.pi * self.radius), 2)
        print(f'The Circumference of Circle = {c}')

r = int(input("Enter Radius of Circle : "))
c = Circle(r)
c.area
c.circumference()
```

Output:

Enter Radius of Circle: 5

The Circumference of Circle = 31.4

Derived Class

 # Example - Derived class

```
class Cal1:
    def Summation(self, a, b):
        return a + b
class Cal2:
    def Multiplication(self, a, b):
        return a * b
class Derived(Cal1, Cal2):
    def Divide(self, a, b):
        return a / b

d = Derived()
print(f"Summation = {d.Summation(10, 20)}")
```

```
print(f"Multiplication = {d.Multiplication(10, 20)}")
print(f"Division = {d.Divide(10, 20)}")
print(f"Derived Class is a subclass of Cal1 class: {issubclass(Derived, Cal1)}")
print(f"Derived Class is a subclass of Cal2 class: {issubclass(Derived, Cal2)}")
print(f"Cal1 Class is a subclass of Cal2 class: {issubclass(Cal1, Cal2)}")
```

Output:

```
Summation = 30
Multiplication = 200
Division = 0.5
Derived Class is a subclass of Cal1 class: True
Derived Class is a subclass of Cal2 class: True
Cal1 Class is a subclass of Cal2 class: False
```

3.5. Static and class methods

3.6. Adding and retrieving dynamic attributes of classes

3.7. Programming using OOPS

3.8. Delegation and container

Python Static Methods Vs Class Method

In this article, we will learn about the difference between a static method and a class method in Python. We will use some built-in functions available in Python and some related custom examples as well. We will compare both methods with examples in this module. Let's first have a quick look over the term decorator, what is a static method, what is a class method, when to use these methods, and compare its working.

What is a Decorator in Python?

Before reading the differences between static and class methods, you must know what is a decorator in Python? Decorators are simple functions. The user can write them, or include them in Python standard library. **Decorators are used for performing logical changes in the**

behavior of other functions. They are an excellent way to reuse code and can they help to separate logic into individual bits. **Python provides decorators to define static and class methods.**

Static Methods

Static methods are methods that are related to a class but do not need to access any class-specific data. There is no need to instantiate an instance because we can simply call this method. Static methods are great for utility functions. They are totally self-contained and only work with data passed in as arguments.

- This method is bound to the class but not to the object of the class.
- This method cannot access or modify class state.
- This method is defined inside a class using **@staticmethod** decorator.
- It does not receive any implicit first argument, neither self nor cls.
- This method returns a static method of the function.

Example: Static Methods

```
class static_example(object):

    #decorator
    @staticmethod
    def fun(arg1, arg2, ...):
        ...
```

Class Methods in Python

Class methods are methods that are related to a class and have access to **all class-specific data**. It uses **@classmethod**, **a built-in function decorator** that gets evaluated after the function is defined. It returns a class method function. It receives the **cls** parameter instead of **self** as the implicit first argument.

- This method is also bound to the class but not to the object of the class.
- This method can access the state of the class, therefore it can modify the class state that will be applicable to all the instances.
- This method is defined inside a class using **@classmethod** decorator.

- It takes **cls** as a parameter that point to the class and not to the instance of the object.

Example: Define Class Method

```
class class_example(object):
    #decorator
   @classmethod
   def func(cls, arg1, arg2, ...):
      ....
```

Working Example of Static and Class Methods

The below example shows how static and class methods work in a class. **Class methods are used for factory purposes**, that's why in the below code @classmethod details() is used to create a class object from **a birth year** instead of **age**. Static methods are utility functions and work on data provided to them in arguments.

```
from datetime import date
class Person:

   #class constructor
   def __init__(self, name, age):
      self.name =   name
      self.age = age

   @classmethod
   def details(cls, name, year):
      return cls(name, date.today().year - year)

   @staticmethod
   def check_age(age):
      return age > 18

#Driver's code
```

```
person1 = Person('Mark', 20)
person2 = Person.details('Rohan', 1992)
print(person1.name, person1.age)
print(person2.name, person2.age)
print(Person.check_age(25))
```

Output:

```
Mark 20
Rohan 29
True
```

Difference between a static method and a class method

Static Method	Class Method
The @staticmethod decorator is used to create a static method.	The @classmethod decorator is used to create a class method.
No specific parameters are used.	It takes cls as the first parameter.
It cannot access or modify the class state.	It can access or modify the class state.
Static methods do not know about the class state. These methods are used to do some utility tasks by taking some parameters.	The class method takes the class as a parameter to know about the state of that class.
Static methods are used to do some utility tasks.	Class methods are used for factory methods.
It contains totally self-contained code.	It can modify class-specific details.

Conclusion

In this article, we learned about decorators in Python, static methods, and class methods. We learned the working of both methods. We saw key differences between the two methods and how a class defines them.

3.6. Adding and retrieving dynamic attributes of classes

How to create data attributes of a class in run-time in python?

The **setattr** used to sets the named attribute on the given object with a specified value.

Example:

```
class Employee:
    pass

emp1 = Employee()
setattr(emp1, 'Name', 'Milind')
setattr(emp1, 'Salary', 12000)
setattr(emp1, 'Age', 40)

emp2 = Employee()
setattr(emp2, 'Name', 'Ankush')
setattr(emp2, 'Salary', 15000)
setattr(emp2, 'Age', 25)
print(f'Employee - 1 Name = {emp1.Name}, Salary = {emp1.Salary} and Age = {emp1.Age}')
print(f'Employee - 2 Name = {emp2.Name}, Salary = {emp2.Salary} and Age = {emp2.Age}')
```

Output:

```
Employee - 1 Name = Milind, Salary = 12000 and Age = 40
Employee - 2 Name = Ankush, Salary = 15000 and Age = 25
```

Example:

Write a Python class named Student with two attributes student_name, marks. Modify the attribute values of the said class and print the original and modified values of the said attributes.

```
class Student:
    student_name='Abhijeet'
    marks=93

print(f"Student Name: {getattr(Student, 'student_name')}")
print(f"Marks: {getattr(Student, 'marks')}")

setattr(Student, 'student_name', input('Enter name to modify: '))
setattr(Student, 'marks', int(input('Enter marks to modify: ')))

print(f"Student Name: {getattr(Student, 'student_name')}")
print(f"Marks: {getattr(Student, 'marks')}")
```

Output:

```
Student Name: Abhijeet Marks: 93
Enter name to modify: Indu
Enter marks to modify: 85
Student Name: Indu Marks: 85
```

Example:

Write a Python class named Student with two attributes student_id, student_name. Add a new attribute student_class and display the entire attribute and their values of the said class. Now remove the student_name attribute and display the entire attribute with values.

```
class Student:
    student_id = '101'
    student_name = 'Ankush Aher'

print("Original attributes and their values of the Student class:")
for attr, value in Student.__dict__.items():
    if not attr.startswith('_'):
        print(f'{attr} -> {value}')
```

```
print("\nAfter adding the student_class, attributes and their values with the said class:")
Student.student_class  = 'MCA-I A'
for attr, value in Student.__dict__.items():
    if not attr.startswith('_'):
        print(f'{attr} -> {value}')

print("\nAfter removing the student_name, attributes and their values from the said class:")
del Student.student_name
#delattr(Student, 'student_name')
for attr, value in Student.__dict__.items():
    if not attr.startswith('_'):
        print(f'{attr} -> {value}')
```

Output:

```
Original attributes and their values of the Student class:
student_id -> 101
student_name -> Ankush Aher

After adding the student_class, attributes and their values with the
said class:
student_id -> 101
student_name -> Ankush Aher
student_class -> MCA-I A

After removing the student_name, attributes and their values from the
said class:
student_id -> 101
student_class -> MCA-I A
```

Example:

Python program to create a class and display the namespace of the Student class.

```
class student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def disply(self):
        print(f'name = {self.name} and age = {self.age}')

for name in student.__dict__:
    print(name)
```

Output:

```
__module__
__init__
disply
__dict__
__weakref__
__doc__
```

Example:

Write a Python program to create an instance of a specified class and display the namespace of the said instance.

```
class Student:
    def __init__(self, student_id, student_name, class_name):
        self.student_id = student_id
        self.student_name = student_name
        self.class_name = class_name
```

```
student = Student('101', 'Ankush', 'MCA-I A')
print(student.__dict__)
```

Output: `{'student_id': '101', 'student_name': 'Ankush', 'class_name': 'MCA-I A'}`

3.8. Delegation and Container

Example of Composition in Python

In composition one of the classes is composed of one or more instance of other classes such class is called as Container class. In other words one class is **container** and other class is **content** and **if you delete the container object then all of its contents objects are also deleted.**

Example:

```
class Salary:
    def __init__(self, pay):
        self.pay = pay

    def get_total(self):
        return (self.pay*12)

class Employee:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus
        self.obj_salary = Salary(self.pay)

    def annual_salary(self):
        return "Total: " + str(self.obj_salary.get_total() + self.bonus)

obj_emp = Employee(600, 500)
print(obj_emp.annual_salary())
```

Output: `Total: 7700`

Example of Aggregation in Python

Aggregation is a week form of composition. If you delete the container object, contents objects can live without container object.

```
class Salary:
    def __init__(self, pay):
        self.pay = pay

    def get_total(self):
        return (self.pay*12)

 class Employee:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def annual_salary(self):
        return "Total: " + str(self.pay.get_total() + self.bonus)

obj_sal = Salary(600)
obj_emp = Employee(obj_sal, 500)
print(obj_emp.annual_salary())
```

Output:

```
Total: 7700
```