Python OOPs Concepts

Like other general purpose languages, python is also an object-oriented language since its beginning. Python is an object-oriented programming language. It allows us to develop applications using an Object Oriented approach. In Python, we can easily create and use classes and objects.

Major principles of object-oriented programming system are given below.

* Object
* Class
* Method
* Inheritance
* Polymorphism
* Data Abstraction
* Encapsulation

Object

The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute \_\_doc\_\_, which returns the doc string defined in the function source code.

Class

The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have an employee class then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.

Syntax

1. **class** ClassName:
2. <statement-1>
3. .
4. .
5. <statement-N>

Method

The method is a function that is associated with an object. In Python, a method is not unique to class instances. Any object type can have methods.

Inheritance

Inheritance is the most important aspect of object-oriented programming which simulates the real world concept of inheritance. It specifies that the child object acquires all the properties and behaviors of the parent object.

By using inheritance, we can create a class which uses all the properties and behavior of another class. The new class is known as a derived class or child class, and the one whose properties are acquired is known as a base class or parent class.

It provides re-usability of the code.

Polymorphism

Polymorphism contains two words "poly" and "morphs". Poly means many and Morphs means form, shape. By polymorphism, we understand that one task can be performed in different ways. For example You have a class animal, and all animals speak. But they speak differently. Here, the "speak" behavior is polymorphic in the sense and depends on the animal. So, the abstract "animal" concept does not actually "speak", but specific animals (like dogs and cats) have a concrete implementation of the action "speak".

Encapsulation

Encapsulation is also an important aspect of object-oriented programming. It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

Data Abstraction

Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonym because data abstraction is achieved through encapsulation.

Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does.

Object-oriented vs Procedure-oriented Programming languages

|  |  |  |
| --- | --- | --- |
| **Index** | **Object-oriented Programming** | **Procedural Programming** |
| 1. | Object-oriented programming is the problem-solving approach and used where computation is done by using objects. | Procedural programming uses a list of instructions to  do computation step by step. |
| 2. | It makes the development and maintenance easier. | In procedural programming, It is not easy to maintain  the codes when the project becomes lengthy. |
| 3. | It simulates the real world entity. So real-world problems can be easily solved through oops. | It doesn't simulate the real world. It works on step  by step instructions divided into small parts  called functions. |
| 4. | It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere. | Procedural language doesn't provide any proper way  for data binding, so it is less secure. |
| 5. | Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc. | Example of procedural languages are: C, Fortran,  Pascal, VB etc. |

Python Class and Objects

As we have already discussed, a class is a virtual entity and can be seen as a blueprint of an object. The class came into existence when it instantiated. Let's understand it by an example.

Suppose a class is a prototype of a building. A building contains all the details about the floor, doors, windows, etc. we can make as many buildings as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.

On the other hand, the object is the instance of a class. The process of creating an object can be called as instantiation.

In this section of the tutorial, we will discuss creating classes and objects in python. We will also talk about how an attribute is accessed by using the class object.

Creating classes in python

In python, a class can be created by using the keyword class followed by the class name. The syntax to create a class is given below.

Syntax

1. **class** ClassName:
2. #statement\_suite

In python, we must notice that each class is associated with a documentation string which can be accessed by using **<class-name>.\_\_doc\_\_**. A class contains a statement suite including fields, constructor, function, etc. definition.

Consider the following example to create a class Employee which contains two fields as Employee id, and name.

The class also contains a function display() which is used to display the information of the Employee.

Example

1. **class** Employee:
2. id = 10;
3. name = "ayush"
4. **def** display (self):
5. **print**(self.id,self.name)

Here, the self is used as a reference variable which refers to the current class object. It is always the first argument in the function definition. However, using self is optional in the function call.

Creating an instance of the class

A class needs to be instantiated if we want to use the class attributes in another class or method. A class can be instantiated by calling the class using the class name.

The syntax to create the instance of the class is given below.

1. <object-name> = <**class**-name>(<arguments>)

The following example creates the instance of the class Employee defined in the above example.

Example

1. **class** Employee:
2. id = 10;
3. name = "John"
4. **def** display (self):
5. **print**("ID: %d \nName: %s"%(self.id,self.name))
6. emp = Employee()
7. emp.display()

**Output:**

ID: 10

Name: ayush

Python Constructor

A constructor is a special type of method (function) which is used to initialize the instance members of the class.

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

Constructor definition is executed when we create the object of this class. Constructors also verify that there are enough resources for the object to perform any start-up task.

Creating the constructor in python

In python, the method \_\_**init**\_\_ simulates the constructor of the class. This method is called when the class is instantiated. We can pass any number of arguments at the time of creating the class object, depending upon \_\_**init**\_\_ definition. It is mostly used to initialize the class attributes. Every class must have a constructor, even if it simply relies on the default constructor.

Consider the following example to initialize the Employee class attributes.

Example

1. **class** Employee:
2. **def** \_\_init\_\_(self,name,id):
3. self.id = id;
4. self.name = name;
5. **def** display (self):
6. **print**("ID: %d \nName: %s"%(self.id,self.name))
7. emp1 = Employee("John",101)
8. emp2 = Employee("David",102)
10. #accessing display() method to print employee 1 information
12. emp1.display();
14. #accessing display() method to print employee 2 information
15. emp2.display();

**Output:**

ID: 101

Name: John

ID: 102

Name: David

Example: Counting the number of objects of a class

1. **class** Student:
2. count = 0
3. **def** \_\_init\_\_(self):
4. Student.count = Student.count + 1
5. s1=Student()
6. s2=Student()
7. s3=Student()
8. **print**("The number of students:",Student.count)

**Output:**

The number of students: 3

Python Non-Parameterized Constructor Example

1. **class** Student:
2. # Constructor - non parameterized
3. **def** \_\_init\_\_(self):
4. **print**("This is non parametrized constructor")
5. **def** show(self,name):
6. **print**("Hello",name)
7. student = Student()
8. student.show("John")

**Output:**

This is non parametrized constructor

Hello John

Python Parameterized Constructor Example

1. **class** Student:
2. # Constructor - parameterized
3. **def** \_\_init\_\_(self, name):
4. **print**("This is parametrized constructor")
5. self.name = name
6. **def** show(self):
7. **print**("Hello",self.name)
8. student = Student("John")
9. student.show()

**Output:**

This is parametrized constructor

Hello John

Python In-built class functions

The in-built functions defined in the class are described in the following table.

|  |  |  |
| --- | --- | --- |
| **SN** | **Function** | **Description** |
| 1 | getattr(obj,name,default) | It is used to access the attribute of the object. |
| 2 | setattr(obj, name,value) | It is used to set a particular value to the specific attribute of an object. |
| 3 | delattr(obj, name) | It is used to delete a specific attribute. |
| 4 | hasattr(obj, name) | It returns true if the object contains some specific attribute. |

Example

1. **class** Student:
2. **def** \_\_init\_\_(self,name,id,age):
3. self.name = name;
4. self.id = id;
5. self.age = age
7. #creates the object of the class Student
8. s = Student("John",101,22)
10. #prints the attribute name of the object s
11. **print**(getattr(s,'name'))
13. # reset the value of attribute age to 23
14. setattr(s,"age",23)
16. # prints the modified value of age
17. **print**(getattr(s,'age'))
19. # prints true if the student contains the attribute with name id
21. **print**(hasattr(s,'id'))
22. # deletes the attribute age
23. delattr(s,'age')
25. # this will give an error since the attribute age has been deleted
26. **print**(s.age)

**Output:**

John

23

True

AttributeError: 'Student' object has no attribute 'age'

Built-in class attributes

Along with the other attributes, a python class also contains some built-in class attributes which provide information about the class.

The built-in class attributes are given in the below table.

|  |  |  |
| --- | --- | --- |
| **SN** | **Attribute** | **Description** |
| 1 | \_\_dict\_\_ | It provides the dictionary containing the information about the class namespace. |
| 2 | \_\_doc\_\_ | It contains a string which has the class documentation |
| 3 | \_\_name\_\_ | It is used to access the class name. |
| 4 | \_\_module\_\_ | It is used to access the module in which, this class is defined. |
| 5 | \_\_bases\_\_ | It contains a tuple including all base classes. |

Example

1. **class** Student:
2. **def** \_\_init\_\_(self,name,id,age):
3. self.name = name;
4. self.id = id;
5. self.age = age
6. **def** display\_details(self):
7. **print**("Name:%s, ID:%d, age:%d"%(self.name,self.id))
8. s = Student("John",101,22)
9. **print**(s.\_\_doc\_\_)
10. **print**(s.\_\_dict\_\_)
11. **print**(s.\_\_module\_\_)

**Output:**

None

{'name': 'John', 'id': 101, 'age': 22}

\_\_main\_\_