Python OOPs Concepts

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming.

The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

OOPs Concepts in Python

* Class
* Objects
* Polymorphism
* Encapsulation
* Inheritance
* Data Abstraction

Python Class

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

To understand the need for creating a class let’s consider an example, let’s say you wanted to track the number of dogs that may have different attributes like breed, and age. If a list is used, the first element could be the dog’s breed while the second element could represent its age. Let’s suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it’s the exact need for classes.

Some points on Python class:

* + Classes are created by keyword class.
  + Attributes are the variables that belong to a class.
  + Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

Class Definition Syntax:

class ClassName:

# Statement-1

.

.

.

# Statement-N

Creating an Empty Class in Python

In the above example, we have created a class named Dog using the class keyword.

class Dog:

pass

Python Objects

The object is an entity that has a state and behaviour associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number 12 is an object, the string “Hello, world” is an object, a list is an object that can hold other objects, and so on. You’ve been using objects all along and may not even realize it.

An object consists of:

**State**: It is represented by the attributes of an object. It also reflects the properties of an object.

**Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.

**Identity**: It gives a unique name to an object and enables one object to interact with other objects.

To understand the state, behavior, and identity let us take the example of the class dog (explained above).

* The identity can be considered as the name of the dog.
* State or Attributes can be considered as the breed, age, or color of the dog.
* The behavior can be considered as to whether the dog is eating or sleeping.

Creating an Object

This will create an object named obj of the class Dog defined above. Before diving deep into objects and classes let us understand some basic keywords that will we used while working with objects and classes.

obj = Dog()

The Python self

1. Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it
2. If we have a method that takes no arguments, then we still have to have one argument.
3. This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

The Python \_\_init\_\_ Method

The \_\_init\_\_ method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

Now let us define a class and create some objects using the self and \_\_init\_\_ method.

Creating a class and object with class and instance attributes

class Dog:

# class attribute

attr1 = "mammal"

# Instance attribute

def \_\_init\_\_(self, name):

self.name = name

# Driver code

# Object instantiation

Rodger = Dog("Rodger")

Tommy = Dog("Tommy")

# Accessing class attributes

print("Rodger is a {}".format(Rodger.\_\_class\_\_.attr1))

print("Tommy is also a {}".format(Tommy.\_\_class\_\_.attr1))

# Accessing instance attributes

print("My name is {}".format(Rodger.name))

print("My name is {}".format(Tommy.name))

Output

Rodger is a mammal

Tommy is also a mammal

My name is Rodger

My name is Tommy

Creating Classes and objects with methods

Here, The Dog class is defined with two attributes:

attr1 is a class attribute set to the value “mammal”. Class attributes are shared by all instances of the class.

\_\_init\_\_ is a special method (constructor) that initializes an instance of the Dog class. It takes two parameters: self (referring to the instance being created) and name (representing the name of the dog).

The name parameter is used to assign a name attribute to each instance of Dog.

The speak method is defined within the Dog class. This method prints a string that includes the name of the dog instance.

The driver code starts by creating two instances of the Dog class: Rodger and Tommy. The \_\_init\_\_ method is called for each instance to initialize their name attributes with the provided names.

The speak method is called in both instances (Rodger.speak() and Tommy.speak()), causing each dog to print a statement with its name.

class Dog:

# class attribute

attr1 = "mammal"

# Instance attribute

def \_\_init\_\_(self, name):

self.name = name

def speak(self):

print("My name is {}".format(self.name))

# Driver code

# Object instantiation

Rodger = Dog("Rodger")

Tommy = Dog("Tommy")

# Accessing class methods

Rodger.speak()

Tommy.speak()

Output

My name is Rodger

My name is Tommy

Note: For more information, refer to Python Classes and Objects

Python Inheritance

Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or

child class and the class from which the properties are being derived is called the base class or parent class. The benefits of inheritance are:

* It represents real-world relationships well.
* It provides the reusability of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
* It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Types of Inheritance

**Single Inheritance:** Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.

**Multilevel Inheritance:** Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.

**Hierarchical Inheritance:** Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.

**Multiple Inheritance:** Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

Inheritance in Python

In the above article, we have created two classes i.e. Person (parent class) and Employee (Child Class). The Employee class inherits from the Person class.

We can use the methods of the person class through the employee class as seen in the display function in the above code. A child class can also modify the behavior of the parent class as seen through the details() method.

class Person(object):

# \_\_init\_\_ is known as the constructor

def \_\_init\_\_(self, name, idnumber):

self.name = name

self.idnumber = idnumber

def display(self):

print(self.name)

print(self.idnumber)

def details(self):

print("My name is {}".format(self.name))

print("IdNumber: {}".format(self.idnumber))

# child class

class Employee(Person):

def \_\_init\_\_(self, name, idnumber, salary, post):

self.salary = salary

self.post = post

# invoking the \_\_init\_\_ of the parent class

Person.\_\_init\_\_(self, name, idnumber)

def details(self):

print("My name is {}".format(self.name))

print("IdNumber: {}".format(self.idnumber))

print("Post: {}".format(self.post))

# creation of an object variable or an instance

a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using

# its instance

a.display()

a.details()

Output

Rahul

886012

My name is Rahul

IdNumber: 886012

Post: Intern

**Python Polymorphism**

Polymorphism simply means having many forms. For example, we need to determine if the given species of birds fly or not, using polymorphism we can do this using a single function.

**Polymorphism in Python**

This code demonstrates the concept of inheritance and method overriding in Python classes.

It shows how subclasses can override methods defined in their parent class to provide specific behavior while still inheriting other methods from the parent class.

class Bird:

def intro(self):

print("There are many types of birds.")

def flight(self):

print("Most of the birds can fly but some cannot.")

class sparrow(Bird):

def flight(self):

print("Sparrows can fly.")

class ostrich(Bird):

def flight(self):

print("Ostriches cannot fly.")

obj\_bird = Bird()

obj\_spr = sparrow()

obj\_ost = ostrich()

obj\_bird.intro()

obj\_bird.flight()

obj\_spr.intro()

obj\_spr.flight()

obj\_ost.intro()

obj\_ost.flight()

Output

There are many types of birds.

Most of the birds can fly but some cannot.

There are many types of birds.

Sparrows can fly.

There are many types of birds.

Ostriches cannot fly.

**Python Encapsulation**

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit.

This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object’s variable can only be changed by an object’s method.

Those types of variables are known as private variables.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

**Encapsulation in Python**

In the above example, we have created the c variable as the private attribute. We cannot even access this attribute directly and can’t even change its value.

class Base:

def \_\_init\_\_(self):

self.a = "GeeksforGeeks"

self.\_\_c = "GeeksforGeeks"

# Creating a derived class

class Derived(Base):

def \_\_init\_\_(self):

# Calling constructor of

# Base class

Base.\_\_init\_\_(self)

print("Calling private member of base class: ")

print(self.\_\_c)

# Driver code

obj1 = Base()

print(obj1.a)

# Uncommenting print(obj1.c) will

# raise an AttributeError

# Uncommenting obj2 = Derived() will

# also raise an AtrributeError as

# private member of base class

# is called inside derived class

Output

GeeksforGeeks

Data Abstraction

It hides unnecessary code details from the user. Also, when we do not want to give out sensitive parts of our code implementation and this is where data abstraction came.

Data Abstraction in Python can be achieved by creating abstract classes.

Object Oriented Programming in Python | Set 2 (Data Hiding and Object Printing)

List of Python Magic Methods

To get the list of magic functions in Python, open cmd or terminal, type python3 to go to the Python console, and type:

>>> dir(int)

Output:

Magic methods in Python

Python’s Magic Methods Guide

Below are the lists of magic functions in Python and their uses.

Initialization and Construction

\_\_new\_\_: To get called in an object’s instantiation.

\_\_init\_\_: To get called by the \_\_new\_\_ method.

\_\_del\_\_: It is the destructor.

Numeric magic methods

\_\_trunc\_\_(self): Implements behavior for math.trunc()

\_\_ceil\_\_(self): Implements behavior for math.ceil()

\_\_floor\_\_(self): Implements behavior for math.floor()

\_\_round\_\_(self,n): Implements behavior for the built-in round()

\_\_invert\_\_(self): Implements behavior for inversion using the ~ operator.

\_\_abs\_\_(self): Implements behavior for the built-in abs()

\_\_neg\_\_(self): Implements behavior for negation

\_\_pos\_\_(self): Implements behavior for unary positive

Arithmetic operators

\_\_add\_\_(self, other): Implements behavior for math.trunc()

\_\_sub\_\_(self, other): Implements behavior for math.ceil()

\_\_mul\_\_(self, other): Implements behavior for math.floor()

\_\_floordiv\_\_(self, other): Implements behavior for the built-in round()

\_\_div\_\_(self, other): Implements behavior for inversion using the ~ operator.

\_\_truediv\_\_(self, other): Implements behavior for the built-in abs()

\_\_mod\_\_(self, other): Implements behavior for negation

\_\_divmod\_\_(self, other): Implements behavior for unary positive

\_\_pow\_\_: Implements behavior for exponents using the \*\* operator.

\_\_lshift\_\_(self, other): Implements left bitwise shift using the << operator.

\_\_rshift\_\_(self, other): Implements right bitwise shift using the >> operator.

\_\_and\_\_(self, other): Implements bitwise and using the & operator.

\_\_or\_\_(self, other): Implements bitwise or using the | operator.

\_\_xor\_\_(self, other): Implements bitwise xor using the ^ operator.

String Magic Methods

\_\_str\_\_(self): Defines behavior for when str() is called on an instance of your class.

\_\_repr\_\_(self): To get called by built-int repr() method to return a machine readable representation of a type.

\_\_unicode\_\_(self): This method to return an unicode string of a type.

\_\_format\_\_(self, formatstr): return a new style of string.

\_\_hash\_\_(self): It has to return an integer, and its result is used for quick key comparison in dictionaries.

\_\_nonzero\_\_(self): Defines behavior for when bool() is called on an instance of your class.

\_\_dir\_\_(self): This method to return a list of attributes of a class.

\_\_sizeof\_\_(self): It return the size of the object.

Comparison magic methods

\_\_eq\_\_(self, other): Defines behavior for the equality operator, ==.

\_\_ne\_\_(self, other): Defines behavior for the inequality operator, !=.

\_\_lt\_\_(self, other): Defines behavior for the less-than operator, <.

\_\_gt\_\_(self, other): Defines behavior for the greater-than operator, >.

\_\_le\_\_(self, other): Defines behavior for the less-than-or-equal-to operator, <=.

\_\_ge\_\_(self, other): Defines behavior for the greater-than-or-equal-to operator, >=.