

Python - OOP Concepts

OOP is an abbreviation that stands for **Object-oriented programming** paradigm. It is defined as a programming model that uses the concept of **objects** which refers to real-world entities with state and behavior. This chapter helps you become an expert in using object-oriented programming support in Python language.

Python is a programming language that supports object-oriented programming. This makes it simple to create and use classes and objects. If you do not have any prior experience with object-oriented programming, you are at the right place. Let's start by discussing a small introduction of Object-Oriented Programming (OOP) to help you.

Procedural Oriented Approach

Early programming languages developed in 50s and 60s are recognized as procedural (or procedure oriented) languages.

A computer program describes procedure of performing certain task by writing a series of instructions in a logical order. Logic of a more complex program is broken down into smaller but independent and reusable blocks of statements called functions.

Every function is written in such a way that it can interface with other functions in the program. Data belonging to a function can be easily shared with other in the form of arguments, and called function can return its result back to calling function.

Prominent problems related to procedural approach are as follows –

- Its top-down approach makes the program difficult to maintain.
- It uses a lot of global data items, which is undesired. Too many global data items would increase memory overhead.
- It gives more importance to process and doesn't consider data of same importance and takes it for granted, thereby it moves freely through the program.
- Movement of data across functions is unrestricted. In real-life scenario where there is unambiguous association of a function with data it is expected to process.



Advertisement

[Learn More](#)

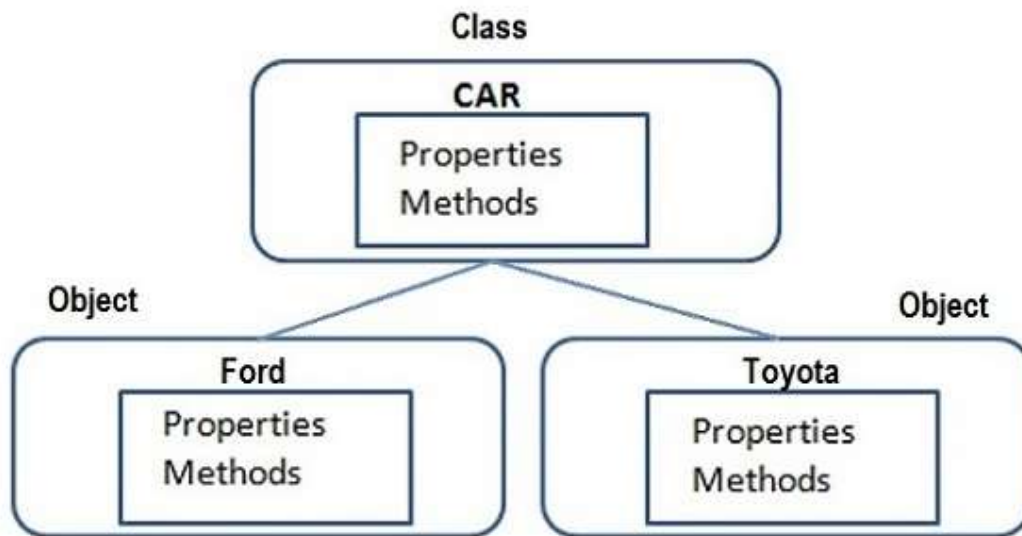
Parallels

直接从 Mac Dock 启动
Windows 应用程序。拥有超过
700 万用户并受到专家好评 -
立享五折优惠!

[Skip Ad](#)

Python - OOP Concepts

In the real world, we deal with and process objects, such as student, employee, invoice, car, etc. Objects are not only data and not only functions, but combination of both. Each real-world object has attributes and behavior associated with it.



Attributes

- Name, class, subjects, marks, etc., of student
- Name, designation, department, salary, etc., of employee
- Invoice number, customer, product code and name, price and quantity, etc., in an invoice
- Registration number, owner, company, brand, horsepower, speed, etc., of car

Each attribute will have a value associated with it. Attribute is equivalent to data.

Behavior

Processing attributes associated with an object.

- Compute percentage of student's marks
- Calculate incentives payable to employee
- Apply GST to invoice value
- Measure speed of car

Behavior is equivalent to function. In real life, attributes and behavior are not independent of each other, rather they co-exist.

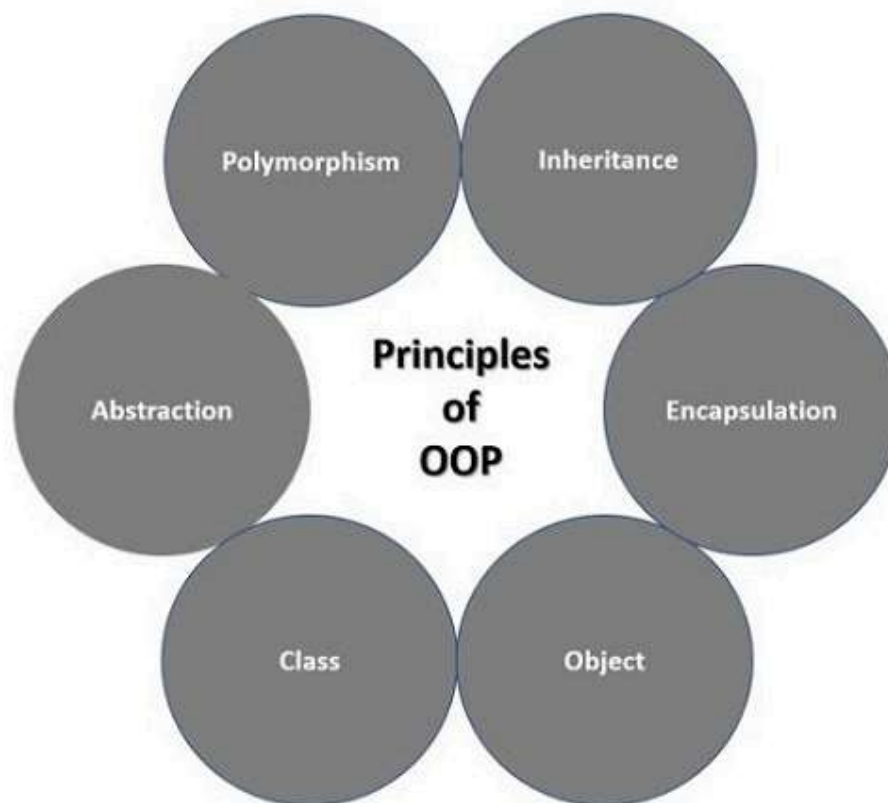
The most important feature of object-oriented approach is defining attributes and their functionality as a single unit called class. It serves as a blueprint for all objects having similar attributes and behavior.

In OOP, class defines what are the attributes its object has, and how is its behavior. Object, on the other hand, is an instance of the class.

Principles of OOPs Concepts

Object-oriented programming paradigm is characterized by the following principles –

- Class
- Object
- Encapsulation
- Inheritance
- Polymorphism



Class & Object

A **class** is an user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

An **object** refers to an instance of a certain class. For example, an object named **obj** that belongs to a class **Circle** is an instance of that class. A unique instance of a data

structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Example

The below example illustrates how to create a class and its object in Python.

```
# defining class
class Smartphone:
    # constructor
    def __init__(self, device, brand):
        self.device = device
        self.brand = brand

    # method of the class
    def description(self):
        return f"{self.device} of {self.brand} supports Android 14"

# creating object of the class
phoneObj = Smartphone("Smartphone", "Samsung")
print(phoneObj.description())
```

On executing the above code, it will display the following output –

```
Smartphone of Samsung supports Android 14
```

Encapsulation

Data members of class are available for processing to functions defined within the class only. Functions of class on the other hand are accessible from outside class context. So object data is hidden from environment that is external to class. Class function (also called method) encapsulates object data so that unwarranted access to it is prevented.

Example

In this example, we are using the concept of encapsulation to set the price of desktop.



```

class Desktop:
    def __init__(self):
        self.__max_price = 25000

    def sell(self):
        return f"Selling Price: {self.__max_price}"

    def set_max_price(self, price):
        if price > self.__max_price:
            self.__max_price = price

# Object
desktopObj = Desktop()
print(desktopObj.sell())

# modifying the price directly
desktopObj.__max_price = 35000
print(desktopObj.sell())

# modifying the price using setter function
desktopObj.set_max_price(35000)
print(desktopObj.sell())

```

When the above code is executed, it produces the following result –

```

Selling Price: 25000
Selling Price: 25000
Selling Price: 35000

```

Inheritance

A software modelling approach of OOP enables extending capability of an existing class to build new class instead of building from scratch. In OOP terminology, existing class is called **base or parent class**, while new class is called **child or sub class**.

Child class inherits data definitions and methods from parent class. This facilitates reuse of features already available. Child class can add few more definitions or redefine a base class function.

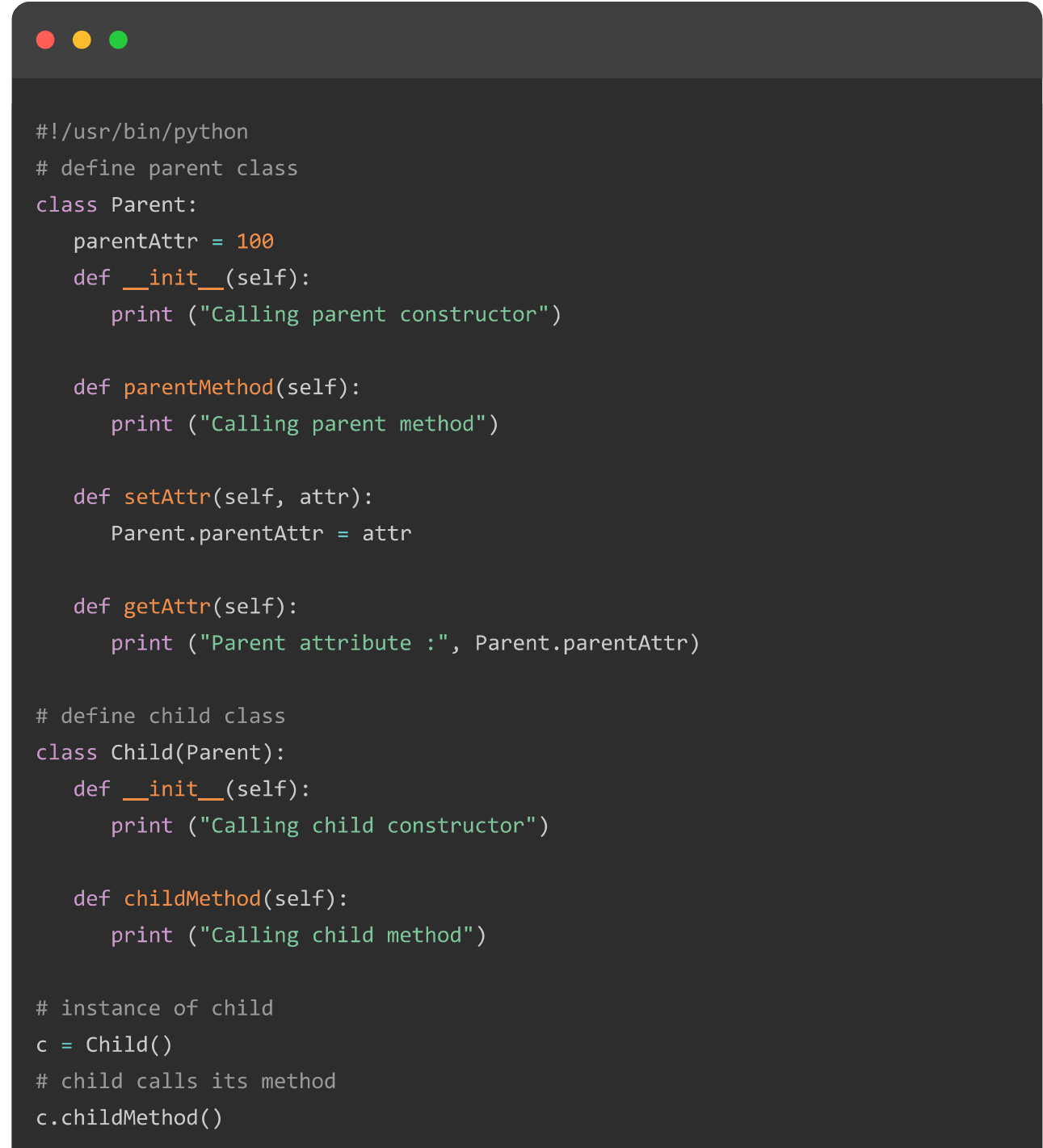
Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name –

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
    'Optional class documentation string'
    class_suite
```

Example

The following example demonstrates the concept of Inheritance in Python –



```
#!/usr/bin/python
# define parent class
class Parent:
    parentAttr = 100
    def __init__(self):
        print ("Calling parent constructor")

    def parentMethod(self):
        print ("Calling parent method")

    def setAttr(self, attr):
        Parent.parentAttr = attr

    def getAttr(self):
        print ("Parent attribute :", Parent.parentAttr)

# define child class
class Child(Parent):
    def __init__(self):
        print ("Calling child constructor")

    def childMethod(self):
        print ("Calling child method")

# instance of child
c = Child()
# child calls its method
c.childMethod()
```

```
# calls parent's method
c.parentMethod()
# again call parent's method
c.setAttr(200)
# again call parent's method
c.getAttr()
```

When the above code is executed, it produces the following result –

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Similar way, you can drive a class from multiple parent classes as follows –

```
class A:          # define your class A
.....

class B:          # define your class B
.....

class C(A, B):    # subclass of A and B
.....
```

You can use `issubclass()` or `isinstance()` functions to check a relationships of two classes and instances.

- The **issubclass(sub, sup)** boolean function returns true if the given subclass **sub** is indeed a subclass of the superclass **sup**.
- The **isinstance(obj, Class)** boolean function returns true if `obj` is an instance of class `Class` or is an instance of a subclass of `Class`

Polymorphism

Polymorphism is a Greek word meaning having multiple forms. In OOP, polymorphism occurs when each sub class provides its own implementation of an abstract method in base class.

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

Example

In this example, we are overriding the parent's method.

```
# define parent class
class Parent:
    def myMethod(self):
        print ("Calling parent method")

# define child class
class Child(Parent):
    def myMethod(self):
        print ("Calling child method")

# instance of child
c = Child()
# child calls overridden method
c.myMethod()
```

When the above code is executed, it produces the following result –

```
Calling child method
```

Base Overloading Methods in Python

Following table lists some generic functionality that you can override in your own classes –

Sr.No.	Method, Description & Sample Call
1	<code>__init__ (self [,args...])</code> Constructor (with any optional arguments) Sample Call : <code>obj = className(args)</code>

2	<code>__del__(self)</code> Destructor, deletes an object Sample Call : <code>del obj</code>
3	<code>__repr__(self)</code> Evaluable string representation Sample Call : <code>repr(obj)</code>
4	<code>__str__(self)</code> Printable string representation Sample Call : <code>str(obj)</code>
5	<code>__cmp__(self, x)</code> Object comparison Sample Call : <code>cmp(obj, x)</code>

Overloading Operators in Python

Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the `__add__` method in your class to perform vector addition and then the plus operator would behave as per expectation –

Example

```
class Vector:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __str__(self):
        return 'Vector (%d, %d)' % (self.a, self.b)

    def __add__(self, other):
        return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print (v1 + v2)
```

When the above code is executed, it produces the following result –

```
Vector(7,8)
```

TOP TUTORIALS

Python Tutorial
Java Tutorial
C++ Tutorial
C Programming Tutorial
C# Tutorial
PHP Tutorial
R Tutorial
HTML Tutorial
CSS Tutorial
JavaScript Tutorial
SQL Tutorial

TRENDING TECHNOLOGIES

Cloud Computing Tutorial
Amazon Web Services Tutorial
Microsoft Azure Tutorial
Git Tutorial
Ethical Hacking Tutorial
Docker Tutorial
Kubernetes Tutorial
DSA Tutorial
Spring Boot Tutorial
SDLC Tutorial
Unix Tutorial

CERTIFICATIONS

Business Analytics Certification
Java & Spring Boot Advanced Certification

Data Science Advanced Certification
Cloud Computing And DevOps
Advanced Certification In Business Analytics
Artificial Intelligence And Machine Learning
DevOps Certification
Game Development Certification
Front-End Developer Certification
AWS Certification Training
Python Programming Certification

COMPILERS & EDITORS

Online Java Compiler
Online Python Compiler
Online Go Compiler
Online C Compiler
Online C++ Compiler
Online C# Compiler
Online PHP Compiler
Online MATLAB Compiler
Online Bash Terminal
Online SQL Compiler
Online Html Editor

[ABOUT US](#) | [OUR TEAM](#) | [CAREERS](#) | [JOBS](#) | [CONTACT US](#) | [TERMS OF USE](#) |
[PRIVACY POLICY](#) | [REFUND POLICY](#) | [COOKIES POLICY](#) | [FAQ'S](#)



Tutorials Point is a leading Ed Tech company striving to provide the best learning material on technical and non-technical subjects.

© Copyright 2025. All Rights Reserved.