



PRIME INTUIT

Finishing School

Object Oriented Programming



What is Object Oriented Programming?

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.

OOP Example



A car could be an *object*
Objects have *attributes* and *methods*.

Car Attributes	Car Methods
Make & Model	Drive
Color	Park
# of Doors	Reverse
# of Seats	Tow

Nouns

Verbs

In object-oriented programming (OOP), you have the flexibility to represent real-world objects like car, animal, person, ATM etc. in your code. In simple words, an object is something that possess some characteristics and can perform certain functions. For example, car is an object and can perform functions like start, stop, drive and brake. These are the function of a car. And the characteristics are color of car, mileage, maximum speed, model year etc.

Functions are called methods in OOP world. Characteristics are attributes (properties). Technically attributes are variables or values related to the state of the object whereas methods are functions which have an effect on the attributes of the object.



What is Object Oriented Programming?

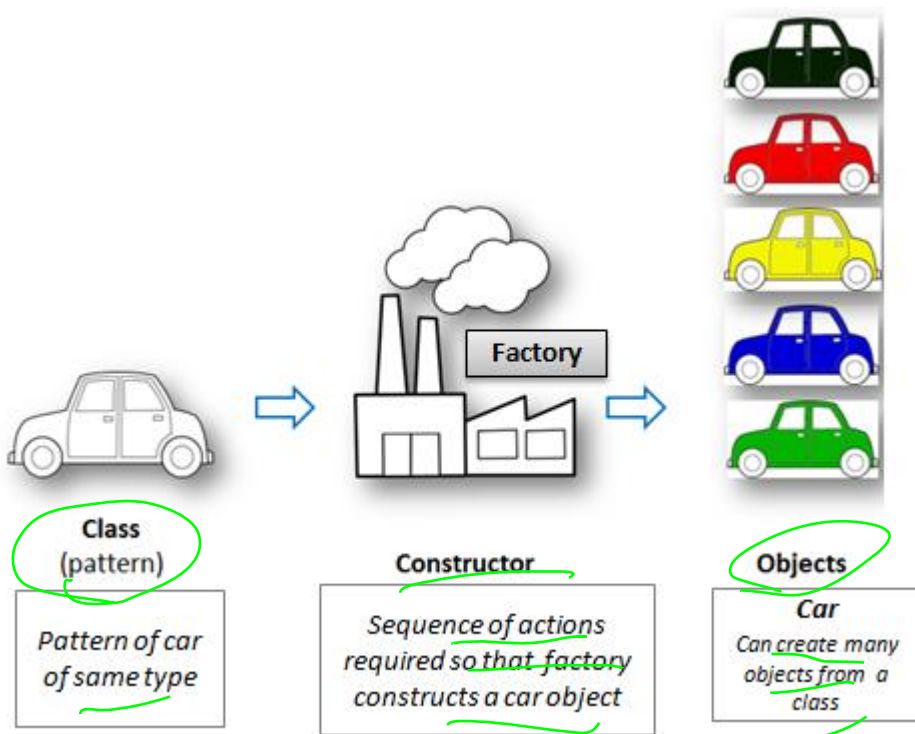
In python, there are mainly 3 programming styles which are Object-Oriented Programming, Functional Programming and Procedural Programming. In simple words, there are 3 different ways to solve the problem in Python. Functional programming is most popular among data scientists as it has performance advantage. OOP is useful when you work with large codebases and code maintainability is very important.

Conclusion : It's good to learn fundamentals of OOP so that you understand what's going behind the libraries you use. If you aim to be a great python developer and want to build Python library, you need to learn OOP (Must!). At the same time there are many data scientists who are unaware of OOP concepts and still excel in their job.



Objects and Classes

Class is a architecture or blue print of the object. It is a proper description of the attributes and methods of a class. For example, design of a car of same type is a class. You can create many objects from a class. Like you can make many cars of the same type from a design of car. 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.



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`

Syntax:

`class ClassName:`

`# Statement-1`

`.`

`.`

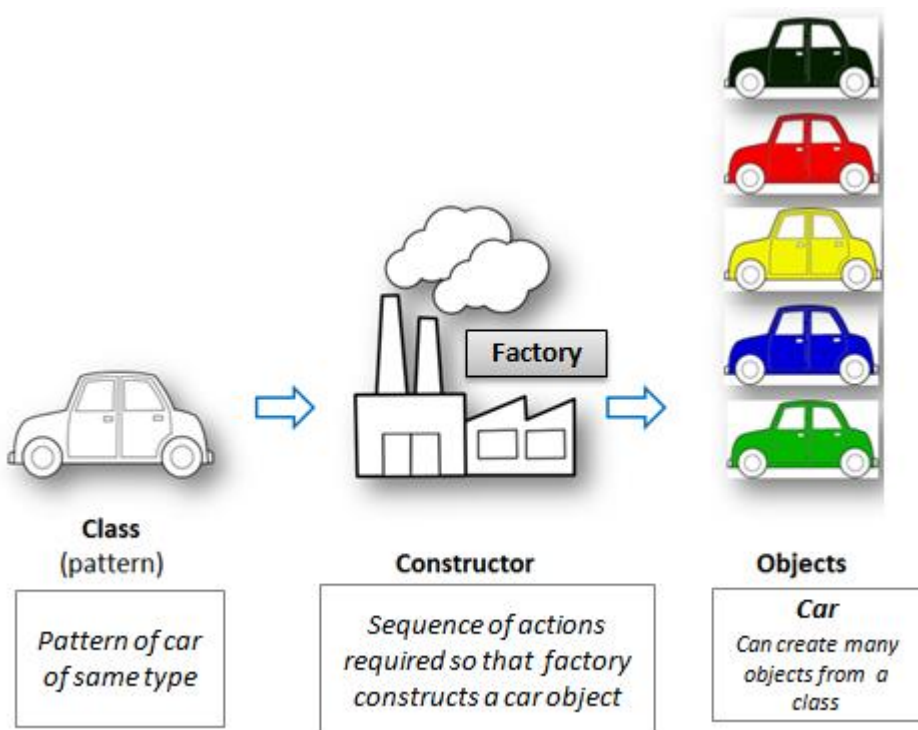
`# Statement-N`



Objects and Classes

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

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



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**.

Syntax:

class car:

obj = car() # obj is **object name that belongs to class car**



self and init

```
class check:
```

```
    def __init__(self):
```

```
        print("Address of self = ",id(self))
```

```
obj = check()
```

```
print("Address of class object = ",id(obj))
```

```
# self and obj is refer to the same object
```

Self represents the instance of the class. By using the “self” keyword we can access the attributes and methods of the class in python. It binds the attributes with the given arguments.

Self 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

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.



Attributes

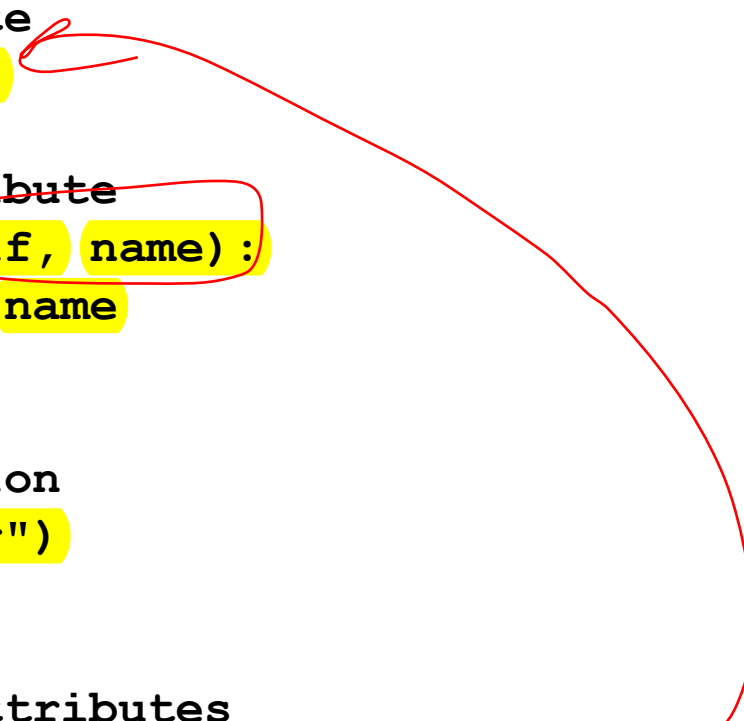
```
# class attribute
attr1 = "mammal"

# 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))
```

A red arrow originates from the 'attr1' attribute in the class definition and points to the 'attr1' attribute access in the 'Rodger.__class__.attr1' line of the driver code.



Attributes

class check:

```
def __init__(self):  
    print("Address of self = ",id(self))
```

```
obj = check()  
print("Address of class object = ",id(obj))
```

self and obj is refer to the same object

Self represents the instance of the class. By using the “self” keyword we can access the attributes and methods of the class in python. It binds the attributes with the given arguments.

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.



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 base class** and the class from which the properties are being derived is called the **base class or parent class**. *could*

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**.



inheritance

Python code to demonstrate how parent constructors
are called.

parent class

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))



inheritance

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))
```

```
N, arr1, arr2 = input().split(" ")
```

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()
```



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.

```
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()
```

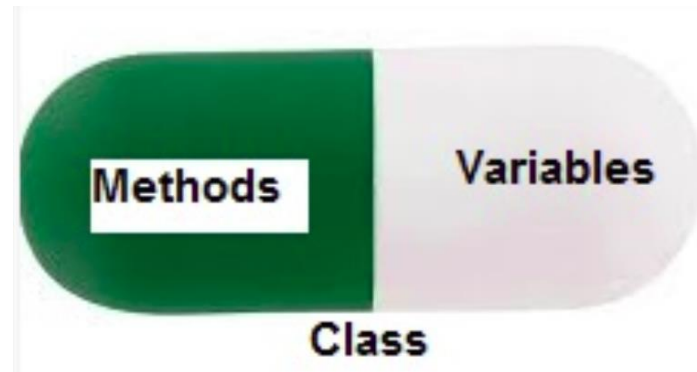
```
obj_ost.intro()
```

```
obj_ost.flight()
```



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

```
# Python program to  
# demonstrate private members
```

```
# Creating a Base class
```

```
class Base:  
    def __init__(self):  
        self.a = "Prime Intuit"  
        self.__c = "Prime Intuit"
```

```
# 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 AttributeError as  
# private member of base class  
# is called inside derived class
```



Bank example

Let's write a simple Python program using OOP concept to perform some simple bank operations like deposit and withdrawal of money.

First of all, define class Bankaccount. This step is followed by defining a function using `__init__`. It is run as soon as an object of a class is instantiated. This `__init__` method is useful to do any initialization you want to do with object, then we have the default argument `self`.

```
# BankAccount class
class Bankaccount:
    def __init__(self):
        self.balance=0
        print("Hello!!! Welcome to the Deposit & Withdrawal Machine")
```



PRIME INTUIT

Finishing School

Bank example

Next, we use a `display` function to display the remaining balance in the account. Then we create a object and call it to get the program executed.

```
# Function to display the amount
```

```
def display(self):
```

```
    print("\n Net Available Balance =", self.balance)
```




Bank example

Use an **if condition** to check whether there is a sufficient amount of money available in the account to process a fund withdrawal.

Function to withdraw the amount

```
def withdraw(self):  
    amount = float(input("Enter amount to be withdrawn: "))  
    if self.balance >= amount:  
        self.balance -= amount  
        print("\n You Withdrew:", amount)  
    else:  
        print("\n Insufficient balance ")
```



Bank example

Code Implementation.

```
class Bank_Account:
    def __init__(self):
        self.balance=0
        print("Hello!!! Welcome to the Deposit & Withdrawal Machine")

    def deposit(self):
        amount=float(input("Enter amount to be Deposited: "))
        self.balance += amount
        print("\n Amount Deposited:",amount)

    def withdraw(self):
        amount = float(input("Enter amount to be Withdrawn: "))
        if self.balance>=amount:
            self.balance-=amount
            print("\n You Withdrew:", amount)
        else:
            print("\n Insufficient balance ")

    def display(self):
        print("\n Net Available Balance=",self.balance)
```

Driver code

creating an object of class

```
s = Bank_Account()
```

Calling functions with that class object

```
s.deposit()
```

```
s.withdraw()
```

```
s.display()
```