**OOPS Concepts in Python**

OOPS in programming stand for **Object Oriented Programming System**. It is a programming paradigm or methodology, to design a program using [classes and objects](https://www.scaler.com/topics/difference-between-class-and-object/) OOPS treats every entity as an object.

Object-oriented programming in Python is centered around objects. Any code written using OOPS is to solve our problem but is represented in the form of Objects. We can create as many objects as we want, for a given class.

So what are objects? **Objects** are anything that has properties and some behaviors. The properties of objects are often referred to as variables of the object, and behaviors are referred to as the functions of the objects. Objects can be real-life or logical.

Suppose, a Pen is a real-life object. The property of a pen includes its color, and type (gel pen or ball pen). And, the behavior of the pen may include that, it can write, draw, etc.

**Some Major Benefits of OOPS Include:**

1. They reduce the redundancy of the code by writing clear and reusable codes (using inheritance).
2. They are easier to visualize because they completely relate to real-world scenarios. For example, the concept of objects, [inheritance](https://www.scaler.com/topics/python/inheritance-in-python/), and abstractions, relate very closely to real-world scenarios(we will discuss them further in this article).
3. Every object in OOPS represent a different part of the code and has its own logic and data to communicate with each other. So, there are no complications in the code.

**Class and Objects in Python**

Suppose you wish to store the number of books you have, you can simply do that by using a variable. Or, say you want to calculate the sum of 5 numbers and store it in a variable, well, that can be done too!

Primitive data structures like numbers, strings, and lists are designed to store simple values in a variable. Suppose, your name, or square of a number, or count of some marbles (say).

But what if you need to store the details of all the Employees in your company? For example, you may try to store every employee in a list, you may later be confused about which index of the list represents what details of the employee(e.g. which is the name field, or the **empID** etc.)

**Example:**

employee1 = ['John Smith', 104120, "Developer", "Dept. 2A"]

employee2 = ['Mark Reeves', 211240, "Database Designer", "Dept. 11B"]

employee3 = ['Steward Jobs', 131124, "Manager", "Dept. 2A"]

Even if you try to store them in a dictionary, after an extent, the whole codebase will be too complex to handle. So, in these scenarios, we use Classes in python.

A **class** is used to create user-defined data structures in Python. Classes define functions, which are termed methods, that describe the behaviors and actions that an object created from a class can perform. OOPS concepts in Python majorly deal with classes and objects.

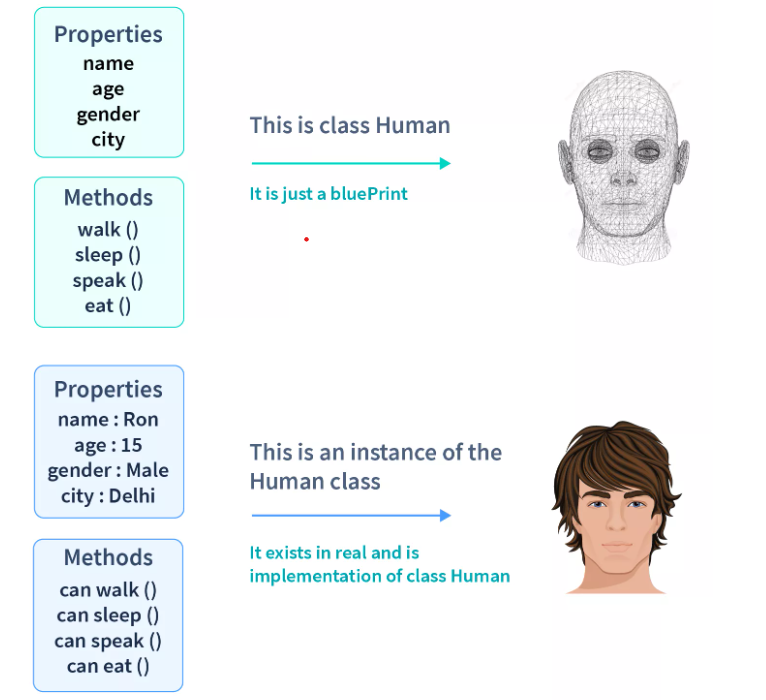
Classes make the code more manageable by avoiding complex codebases. It does so, by creating a blueprint or a design of how anything should be defined. It defines what properties or functions, any object which is derived from the class should have.

**IMPORTANT:**

A class just defines the structure of how anything should look. It does not point to anything or anyone in particular. **For example,** say, HUMAN is a class, which has suppose -- **name, age, gender, city**. It does not point to any specific HUMAN out there, but yes, it explains the properties and functions any HUMAN should or any object of class HUMAN should have.

An instance of a class is called the **object**. It is the implementation of the class and exists in real.

An **object** is a collection of data (variables) and methods (functions) that access the data. It is the real implementation of a class.



Consider this example, here Human is a class - It is just a blueprint that defines how Human should be, and not a real implementation. You may say that "Human" class just exists logically.

However, "Ron" is an object of the Human class (please refer to the image given above for understanding). That means, Ron is created by using the blueprint of the Human class, and it contains the real data. "Ron" exists physically, unlike "Human" (which just exists logically). He exists in **real**, and implements all the **properties** of the class Human, such as, *Ron have a name, he is 15 years old, he is a male, and lives in Delhi*. Also, Ron implements all the **methods** of Human class, suppose, *Ron can walk, speak, eat, and sleep*.

And many humans can be created using the blueprint of class Human. Such as, we may create 1000s of more humans by referring to the blueprint of the class Human, using objects.

When a class is defined, only the blueprint of the object is created, and no memory is allocated to the class. **Memory allocation occurs only when the object or instance is created.** The object or instance contains real data or information.

**How to Define a Class in Python?**

[Classes in Python](https://www.scaler.com/topics/python/class-in-python/) can be defined by the keyword class, which is followed by the name of the class and a colon.

**Syntax:**

class Human:

pass

Indented code below the class definition is considered part of the class body.

**'pass'** is commonly used as a placeholder, in the place of code whose implementation we may skip for the time being. "pass" allows us to run the code without throwing an error in Python.

**What is an \_*init*\_ Method?**

The properties that all Human objects must have been defined in a method called [**init**()](https://www.scaler.com/topics/init-function-in-python/). Every time a new Human object is created, \_\_init\_\_() sets the initial state of the object by assigning the values we provide inside the object’s properties. That is, \_\_init\_\_() initializes each new instance of the class.

\_\_init\_\_() can take any number of parameters, but the first parameter is always a variable called [self](https://www.scaler.com/topics/self-in-python/).

The self parameter is a **reference to the current instance of the class**. It means, the self parameter points to the address of the current object of a class, allowing us to access the data of its(the object's) variables.

So, even if we have **1000 instances** (objects) of a class, we can always get each of their individual data due to this self because it will point to the address of that particular object and return the respective value.

**Note:**

We can use any name in place of self, but it has to be the first parameter of any function in the class.

Let us see, how to define \_\_init\_\_() in the Human class:

**Code:**

class Human:

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

self.name = name

self.age = age

self.gender = gender

In the body of .\_\_init\_\_(), we are using the self variable 3 times, for the following:

self.name = 'name' creates an attribute called name and assigns to it the value of the name parameter.

self.age = age attribute is created and assigned to the value of age parameter passed.

self.gender = gender attribute is created and assigned to the value of gender parameter passed.

There are **2 types of attributes** in Python:

**1. Class Attribute:**

These are the variables that are the same for all instances of the class. They do not have new values for each new instance created. They are defined just below the class definition.

**Code:**

class Human:

*#class attribute*

species = "Homo Sapiens"

Here, the species will have a fixed value for any object we create.

**2. Instance Attribute:**

Instance attributes are the variables that are defined inside of any function in class. Instance attributes have different values for every instance of the class. These values depend upon the value we pass while creating the instance.

**Code:**

class Human:

*#class attribute*

species = "Homo Sapiens"

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

self.name = name

self.age = age

self.gender = gender

Here, name, age, and gender are the instance attributes. They will have different values for new instances of the class.

For properties that should have a similar value per instance of a class, use **class attributes**. For properties that differ per instance, use **instance attributes**.

**Creating an Object in Class**

When we create a new object from a class, it is called **instantiating an object**. An object can be instantiated by the class name followed by the parentheses. We can assign the object of a class to any variable.

**Syntax:**

x = ClassName()

As soon as an object is instantiated, memory is allocated to them. So, if we compare 2 instances of the same class using '==', it will return false(because both will have different memory assigned).

Suppose, we try to create objects of our Human class, then we also need to pass the values for name, age, and gender.

**Code:**

class Human:

*#class attribute*

species = "Homo Sapiens"

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

self.name = name

self.age = age

self.gender = gender

x = Human("Ron", 15, "Male")

y = Human("Miley", 22, "Female")

Here, we have created 2 objects of the class Human passing all the required arguments.

**Warning:** If we do not pass the required arguments, it will throw a **TypeError:** *TypeError: init() missing 3 required positional arguments: 'name', 'age', and 'gender'.*

Let us now see, how to access those values using objects of the class. We can access the values of the instances by using dot notation.

**Code:**

class Human:

*#class attribute*

species = "Homo Sapiens"

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

self.name = name

self.age = age

self.gender = gender

x = Human("Ron", 15, "Male")

y = Human("Miley", 22, "Female")

print(x.name)

print(y.name)

**Output**:

Ron

Miley

So, we find that we can access the instance and class attributes just by using the dot operator.

**Code:**

class Human:

species = "Homo Sapiens"

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

self.name = name

self.age = age

self.gender = gender

*# x and y are instances of class Human*

x = Human("Ron", 15, "male")

y = Human("Miley", 22, "female")

print(x.species) *# species are class attributes, hence will have same value for all instances*

print(y.species)

*# name, gender and age will have different values per instance, because they are instance attributes*

print(f"Hi! My name is {x.name}. I am a {x.gender}, and I am {x.age} years old")

print(f"Hi! My name is {y.name}. I am a {y.gender}, and I am {y.age} years old")

**Output:**

Homo Sapiens

Homo Sapiens

Hi! My name is Ron. I am a male, and I am 15 years old

Hi! My name is Miley. I am a female, and I am 22 years old

In the above example, we have our class attributes values same "Homo Sapiens", but the instance attributes values are different as per the value we passed while creating our object.

However, we can change the value of class attributes, by assigning classname.classAttribute with any new value.

**Code:**

class Human:

*#class attribute*

species = "Homo Sapiens"

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

self.name = name

self.age = age

self.gender = gender

Human.species = "Sapiens"

obj = Human("Brek",11,"male")

print(obj.species)

**Output**:

Sapiens

**Instance Methods**

An instance method is a function defined within a class that can be called only from instances of that class. Like init(), an instance method's first parameter is always self.

Let's take an example and implement some functions can class Human can perform --

**Code:**

class Human:

*#class attribute*

species = "Homo Sapiens"

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

self.name = name

self.age = age

self.gender = gender

*#Instance Method*

def speak(self):

return f"Hello everyone! I am {self.name}"

*#Instance Method*

def eat(self, favouriteDish):

return f"I love to eat {favouriteDish}!!!"

x = Human("Ciri",18,"female")

print(x.speak())

print(x.eat("momos"))

**Output**:

Hello everyone! I am Ciri

I love to eat momos!!!

This Human class has two instance methods:

1. **speak():** It returns a string displaying the name of the Human.
2. **eat():** It has one parameter "favouriteDish" and returns a string displaying the favorite dish of the Human.

Having gained a thorough knowledge of what Python classes, objects, and methods are, it is time for us to turn our focus toward the OOP core principles, upon which it is built.

**Fundamentals of OOPS in Python**

There are four fundamental concepts of Object-oriented programming –

1. Inheritance
2. Encapsulation
3. Polymorphism
4. Data abstraction

Let us now look into each of the OOPS concepts in Python deeply.

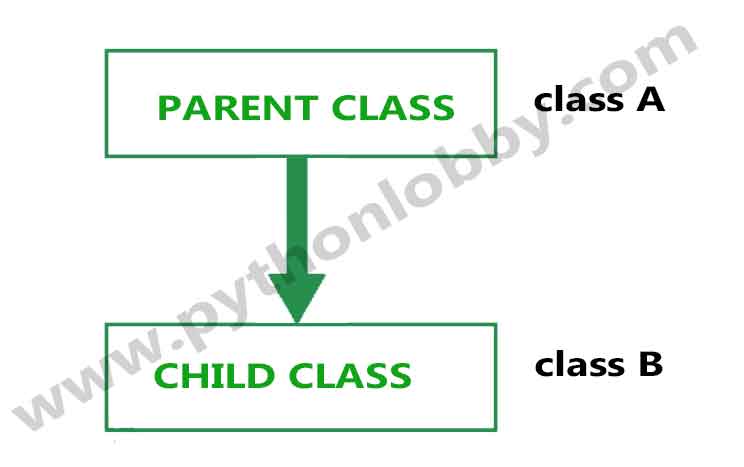
**Inheritance**

**What is Inheritance in Python:**Inheritance in python programming is the concept of deriving a new class from an existing class. Using the concept of inheritance we can inherit the properties of the existing class to our new class. The new derived class is called the child class and the existing class is called the parent class. .

Advantages of Inheritance

* **Code Reusability:**It improves code **reusability**. We don’t need to write the same code again and again. Using inheritance, we can inherit the features of other classes and also add more features to the derived class.
* **Reduces the Programmers Efforts:**Programmers do not need to write the same code and logic. Hence it reduces the efforts of programmers.
* **Readability:**By implementing concepts of inheritance, the program looks more concise and structured. Which makes it easy to read**.**This way inheritance also improves the readability of code.

The diagram to understand what inheritance actually is?



In the above flow diagram class A (parent class) and class B (child class) is there. Class B (child class) is derived from class A. Class B holds all the properties of class A.

**Syntax**to implement inheritance:

class parent\_class:

#parent\_class members

pass

class child\_class(parent\_class):

#child\_class members

pass

obj = child\_class()

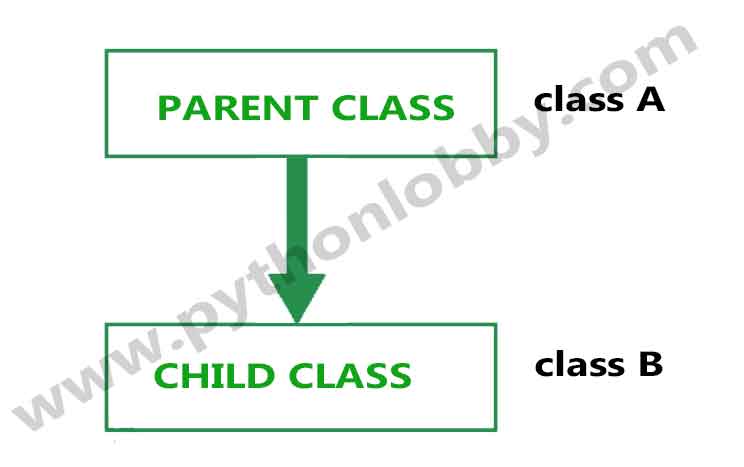
Types of Inheritance in Python Programming

**Types of inheritance:**There are five types of inheritance in python programming:

**1).** Single inheritance  
**2).** Multiple inheritances  
**3).** Multilevel inheritance  
**4).** Hierarchical inheritance  
**5).** Hybrid inheritance

**(i). Single inheritance:**When child class is derived from only one parent class. This is called single inheritance. The example we did above is the best example for single inheritance in python programming.

**Flow Diagram of single inheritance in python programming**



**Syntax of single inheritance:**

#syntax\_of\_single\_inheritance

class class1: #parent\_class

pass

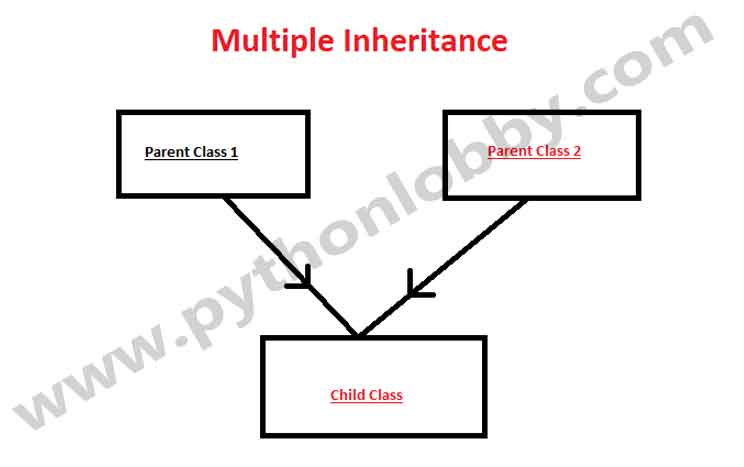
class class2(class1): #child\_class

pass

obj\_name = class2()

**(ii). Multiple Inheritance:**When child class is derived or inherited from more than one parent class. This is called multiple inheritance. In multiple inheritance, we have two parent classes/base classes and one child class that inherits both parent classes properties.

Diagram of Multiple Inheritance



Syntax of multiple inheritance:

#syntax\_of\_multiple\_inheritance

class parent\_1:

pass

class parent\_2:

pass

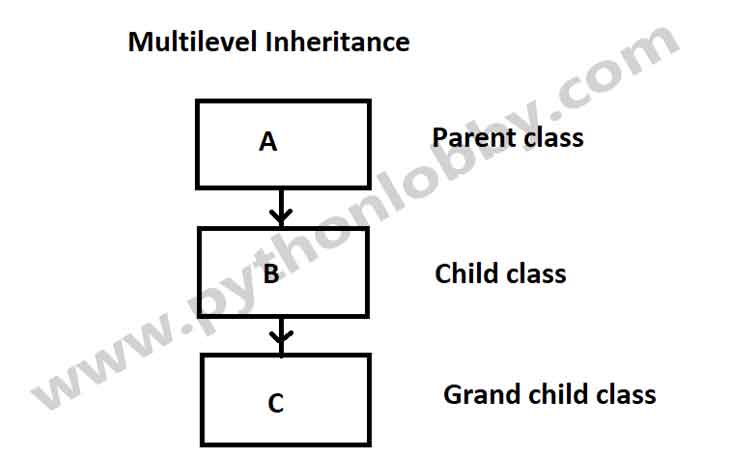
class child(parent\_1,parent\_2):

pass

obj = child()

**(iii). Multilevel Inheritance:**In multilevel inheritance, we have one parent class and child class that is derived or inherited from that parent class. We have a grand-child class that is derived from the child class. See the below-given flow diagram to understand more clearly.

**The flow of Multilevel Inheritance in Python**



Here A is the parent class for class B and class B is acting as the parent class for grand-child class C.

**Syntax**of multilevel inheritance:

#Syntax\_of\_multilevel\_inheritance

class A:

pass

class B(A):

pass

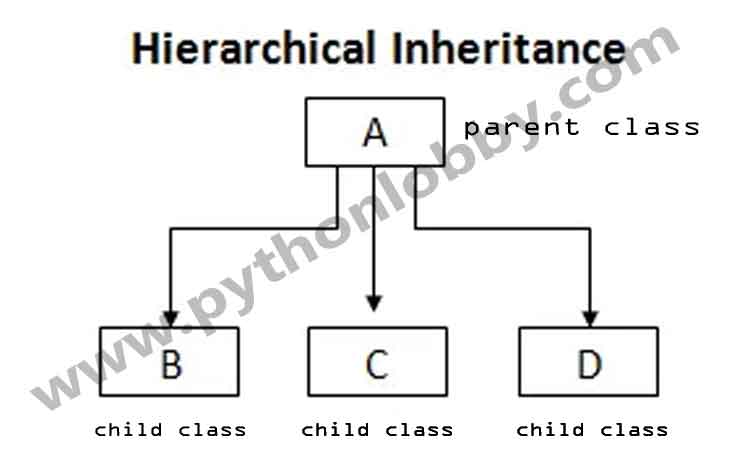
class C(B):

pass

obj = C()

**4). Hierarchical inheritance:**When we derive or inherit more than one child class from one(same) parent class. Then this type of inheritance is called hierarchical inheritance.

**Flow Diagram of Hierarchical Inheritance in Python Programming**



Syntax of Hierarchical Inheritance:

#syntax\_of\_hierarchical\_inheritance

class A: #parent\_class

pass

class B(A): #child\_class

pass

class C(A): #child\_class

pass

class D(A): #child\_class

pass

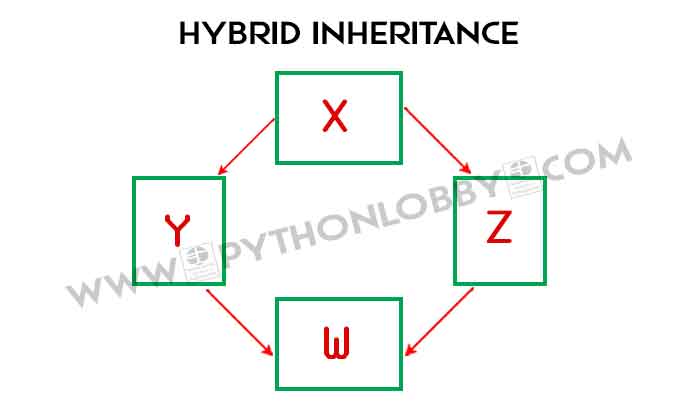
obj\_1 = B() #Object\_creation

obj\_2 = C()

obj\_3 = D()

**5). Hybrid Inheritance:**Hybrid inheritance satisfies more than one form of inheritance ie. It may be consists of all types of inheritance that we have done above. It is not wrong if we say **Hybrid Inheritance** is the combinations of simple, multiple, multilevel and hierarchical inheritance. This type of inheritance is very helpful if we want to use concepts of inheritance without any limitations according to our requirements.

**Flow Diagram of Hybrid Inheritance in Python Programming**



Syntax of Hybrid Inheritance:

#Syntax\_Hybrid\_inheritance

class PC:

pass

class Laptop(PC):

pass

class Mouse(Laptop):

pass

class Student3(Mouse, Laptop):

pass

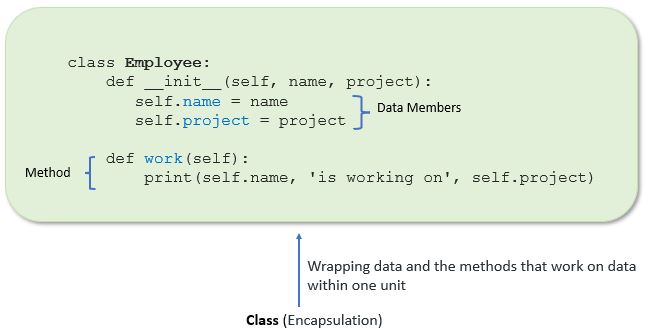
# Driver's code

obj = Student3()

**Note:**There is no sequence in Hybrid inheritance that which class will inherit which particular class. You can use it according to your requirements.

**Encapsulation**

[Encapsulation](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)) in Python describes the concept of **bundling data and**[**methods**](https://pynative.com/python-instance-methods/)**within a single unit**. So, for example, when you create a [class](https://pynative.com/python-classes-and-objects/), it means you are implementing encapsulation. A class is an example of encapsulation as it binds all the data members ([instance variables](https://pynative.com/python-instance-variables/)) and methods into a single unit.

Implement encapsulation using a class

**Example**:

In this example, we create an Employee class by defining employee attributes such as name and salary as an instance variable and implementing behavior using work() and show() instance methods.

**class** Employee:

# constructor

**def** \_\_init\_\_(self, name, salary, project):

# data members

self.name = name

self.salary = salary

self.project = project

# method

# to display employee's details

**def** show(self):

# accessing public data member

**print**("Name: ", self.name, 'Salary:', self.salary)

# method

**def** work(self):

**print**(self.name, 'is working on', self.project)

# creating object of a class

emp = Employee('Jessa', 8000, 'NLP')

# calling public method of the class

emp.show()

emp.work()

[**Run**](https://pynative.com/online-python-code-editor-to-execute-python-code/)

**Output**:

Name: Jessa Salary: 8000

Jessa is working on NLP

Using encapsulation, we can hide an object’s internal representation from the outside. This is called information hiding.

Also, encapsulation allows us to restrict accessing variables and methods directly and prevent accidental data modification by creating private data members and methods within a class.

## Polymorphism

The literal meaning of polymorphism is the condition of occurrence in different forms.

Polymorphism is a very important concept in programming. It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.

## Python has four ways to implement polymorphism

### Duck Typing

In **duck** typing language, objects are typed based on the interface to which they adhere rather than their class. **Duck** typing is a way to check if an object has the methods needed for a certain task in Python. If it does, then the program can execute the task, regardless of the type of object.

### Method Overloading

Method overloading is a feature that enables the same method name to be used with various arguments. When a method is invoked with its arguments specified, it is really invoked in its most precise form. As a result, the code becomes more readable and has less duplication.

### Operator Overloading

By linking an operator to one or more classes, developers can change how different operators work. Because of this polymorphism, programmers may reuse the same operators across a wide range of scenarios.

### Method Overriding

With the help of the method overriding feature, a subclass may change the way a method defined by its parent class operates. This is done by declaring the same method name with the same arguments in the subclass as in the superclass. By overriding methods, developers can customize a sub-class for special tasks.

## Example of implementing polymorphism

### Ducking type

**Duck** typing is a programming style that allows an object to be used in the same way as another object, as long as it has the same methods and properties. In other words, the object’s type is not important; only the methods and properties it has are important.

Code:

*class* *Duck*:

*def* *quack*(self):

        print("Quack quack!")

*class* *Goose*:

*def* *quack*(self):

        print("Honk honk!")

*def* *make\_animal\_sound*(animal):

    animal.quack()

duck = Duck()

goose = Goose()

make\_animal\_sound(duck) *# Outputs: "Quack, quack!"*

make\_animal\_sound(goose) *# Outputs: "Honk honk!"*

In this example, the **make\_animal\_sound()** function takes an animal object as an argument and calls the **quack()** method on it. The Duck and Goose classes both have a **quack()** method, so they can be used with the **make\_animal\_sound()** function. This is an example of duck typing because the object’s type (Duck or Goose) is not important; only the presence of the **quack()** method is important.

### Method Overloading

Using the programming concept of “method overloading,” a class may have numerous methods with the same name but with distinct arguments. When a method is called, the compiler selects which version of the method to run depending on the amount and type of arguments given.But Python does not support method overloading.

Code:

*class* *Calculator*:

*def* *add*(self, a, b):

*return* a + b

*def* *add*(self, a, b, c):

*return* a + b + c

calculator = Calculator()

print(calculator.add(1, 2))   *# error*

print(calculator.add(1, 2, 3)) *# Outputs: 6*

In this example, the Calculator class contains two **add()** methods: one that accepts two parameters and one that takes three arguments. When the **add()** method is invoked with two parameters, it throws an error.Interpreter calls only the last defined add() function.

### Operator Overloading

Using a programming method known as “operator overloading,” familiar operators (such as +, -, \*, etc.) may take on several meanings in various settings.

Let's take an example:

**Example 1: Polymorphism in addition operator**

We know that the + operator is used extensively in Python programs. But, it does not have a single usage.

For integer data types, + operator is used to perform arithmetic addition operation.

num1 = 1

num2 = 2

print(num1+num2)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Hence, the above program outputs 3.

Similarly, for string data types, + operator is used to perform concatenation.

str1 = "Python"

str2 = "Programming"

print(str1+" "+str2)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

As a result, the above program outputs Python Programming.

Here, we can see that a single operator + has been used to carry out different operations for distinct data types. This is one of the most simple occurrences of polymorphism in Python.

### Method Overriding

Overriding a method in a superclass allows a subclass to change how that method is implemented. This means that the subclass implementation of the method will be used whenever the method is invoked on an instance of the subclass rather than the superclass implementation.

Code:

*class* *Animal*:

*def* *make\_sound*(self):

print("Some generic animal sound.")

*class* *Dog*(Animal):

*def* *make\_sound*(self):

print("Bark bark!")

*class* *Cat*(Animal):

*def* *make\_sound*(self):

print("Meow meow!")

dog = Dog()

cat = Cat()

dog.make\_sound() *# Outputs: "Bark bark!"*

cat.make\_sound() *# Outputs: "Meow meow!"*

In this example, the **Animal** class has a **make\_sound()** method that prints a generic animal sound. The **Dog** and **Cat** classes are subclasses of **Animal** and both have their own version of the **make\_sound()** method that prints a specific sound for each type of animal. When the **make\_sound()** method is called on a “Dog” or “Cat” object, the subclass’s version of the method is executed, which overrides the superclass’s version. This is an instance of method overriding, in which a subclass provides an alternative to a method provided by its superclass.

**Abstraction**

We use television to watch shows, news or movies, etc. We use the TV remote to switch the TV ON or OFF, switch to different channels, and raise or lower the volume. The TV user only knows he/she may use the buttons on the remote to do it. What they don’t know is how all this is happening internally, for example how the TV sensor is capturing signals from the TV remote and then how it is processing the received signals to perform the required action of changing the channel, etc. All the internal functionality is hidden, as for the user it might not be necessary for them to know how that is happening.

The example we saw above is one of the examples of abstraction in real life. In object-oriented programming, we shall call it ‘Data Abstraction’. Let us define data abstraction:

**The process by which data and functions are defined in such a way that only essential details can be seen and unnecessary implementations are hidden is called Data Abstraction.**

The main focus of data abstraction is to separate the interface and the implementation of the program.

**Data Abstraction in Python**

Data Abstraction in Python can be achieved through creating abstract classes and inheriting them later. Before discussing what abstract classes are, let us have a brief introduction of inheritance.

Inheritance in OOP is a way through which one class inherits the attributes and methods of another class. The class whose properties and methods are inherited is known as the Parent class. And the class that inherits the properties from the parent class is the Child class/subclass.

The basic syntax to implement [inheritance in Python](https://www.scaler.com/topics/python/inheritance-in-python/) is:

class parent\_class:

body of parent class

class child\_class( parent\_class):

body of child class

Let us now talk about abstract classes in python:

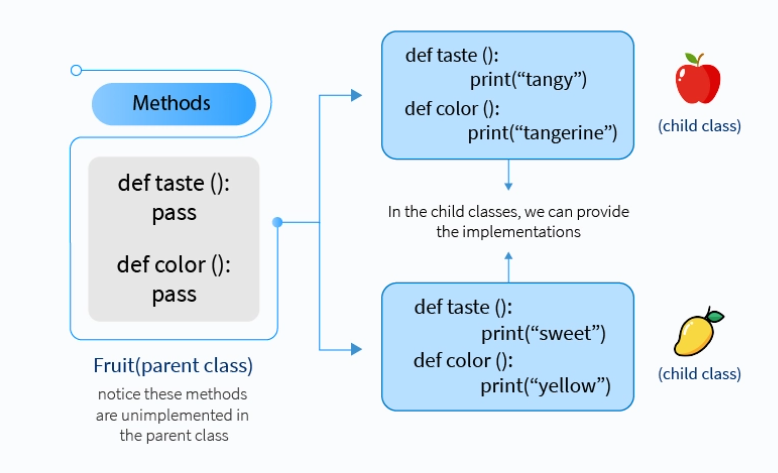
**Abstract Classes in Python**

**Abstract Class:** The classes that cannot be instantiated. This means that we cannot create objects of an abstract class and these are only meant to be inherited. Then an object of the derived class is used to access the features of the base class. These are specifically defined to lay a foundation of other classes that exhibit common behavior or characteristics.

The abstract class is an [interface](https://www.scaler.com/topics/interface-in-python/). Interfaces in OOP enable a class to inherit data and functions from a base class by extending it. In Python, we use the NotImplementedError to restrict the instantiation of a class. Any class having this error inside method definitions cannot be instantiated.

We can understand that an abstract class just serves as a template for other classes by defining a list of methods that the classes must implement. To use such a class, we must derive them keeping in mind that we would either be using or overriding the features specified in that class.

Consider an example where we create an abstract class Fruit. We derive two classes Mango and Orange from the Fruit class that implement the methods defined in this class. Here the Fruit class is the parent abstract class and the classes Mango and Apple become the sub/child classes. We won’t be able to access the methods of the Fruit class by simply creating an object, we will have to create the objects of the derived classes: Mango and Apple to access the methods.



**Why use Abstract Base Class?**

Defining an Abstract Base Class lets us create a common Application Programming Interface (API) for multiple subclasses. It is useful while working in large teams and code-bases so that all of the classes need not be remembered and also be provided as library by third parties.

**Working of Abstract Class**

Unlike other high-level programming languages, Python does not provide the abstract class itself. To achieve that, we use the abc module of Python, which provides the base and necessary tools for defining Abstract Base Classes (ABC). ABCs give the feature of virtual subclasses, which are classes that don’t inherit from a class and can still be recognized by

isinstance()

and

issubclass()

We can create our own ABCs with this module.

from abc import ABC

class MyABC(ABC):

pass

The abc module provides the metaclass ABCMeta for defining ABCs and a helper class ABC to alternatively define ABCs through inheritance. The abc module also provides the @abstractmethod decorator for indicating abstract methods.

ABC is defined in a way that the abstract methods in the base class are created by decorating with the @abstractmethod keyword and the concrete methods are registered as implementations of the base class.

**Concrete Methods in Abstract Base Class in Python**

We now know that we use abstract classes as a template for other similarly characterized classes. Using this, we can define a structure, but do not need to provide complete implementation for every method, such as:

from abc import ABC, abstractmethod

class MyClass(ABC):

@abstractmethod

def mymethod(self):

*#empty body*

pass

The methods where the implementation may vary for any other subclass are defined as abstract methods and need to be given an implementation in the subclass definition.

On the other hand, there are methods that have the same implementation for all subclasses as well. There are characteristics that exhibit the properties of the abstract class and so, must be implemented in the abstract class itself. Otherwise, it will lead to repetitive code in all the inherited classes. These methods are called concrete methods.

from abc import ABC, abstractmethod

class Parent(ABC):

*#common function*

def common\_fn(self):

print('In the common method of Parent')

@abstractmethod

def abs\_fn(self): *#is supposed to have different implementation in child classes*

pass

class Child1(Parent):

def abs\_fn(self):

print('In the abstract method of Child1')

class Child2(Parent):

def abs\_fn(self):

print('In the abstract method of Child2')

An abstract class can have both abstract methods and concrete methods.

We can now access the concrete method of the abstract class by instantiating an object of the child class. We can also access the abstract methods of the child classes with it. Interesting points to keep in mind are:

* We always need to provide an implementation of the abstract method in the child class even when implementation is given in the abstract class.
* A subclass must implement all abstract methods that are defined in the parent class otherwise it results in an error.

**Examples of Data Abstraction**

Let us take some examples to understand the working of abstract classes in Python. Consider the Animal parent class and other child classes derived from it.

from abc import ABC,abstractmethod

class Animal(ABC):

*#concrete method*

def sleep(self):

print("I am going to sleep in a while")

@abstractmethod

def sound(self):

print("This function is for defining the sound by any animal")

pass

class Snake(Animal):

def sound(self):

print("I can hiss")

class Dog(Animal):

def sound(self):

print("I can bark")

class Lion(Animal):

def sound(self):

print("I can roar")

class Cat(Animal):

def sound(self):

print("I can meow")

Our abstract base class has a concrete method [sleep()](https://www.scaler.com/topics/sleep-in-python/) that will be the same for all the child classes. So, we do not define it as an abstract method, thus saving us from code repetition. On the other hand, the sounds that animals make are all different. For that purpose, we defined the sound() method as an abstract method. We then implement it in all child classes.

Now, when we instantiate the child class object, we can access both the concrete and the abstract methods.

c = Cat()

c.sleep()

c.sound()

c = Snake()

c.sound()

This will give the output as:

I am going to sleep in a while

I can meow

I can hiss

Now, if we want to access the sound() function of the base class itself, we can use the object of the child class, but we will have to invoke it through [super()](https://www.scaler.com/topics/super-in-python/).

class Rabbit(Animal):

def sound(self):

super().sound()

print("I can squeak")

c = Rabbit()

c.sound()

This will produce the following output:

This function is for defining the sound by any animal

I can squeak

If we do not provide any implementation of an abstract method in the derived child class, an error is produced. Notice, even when we have given implementation of the sound() method in the base class, not providing an implementation in the child class will produce an error.

class Deer(Animal):

def sound(self):

pass

c = Deer()

c.sound()

c.sleep()

This will produce the following error:

Traceback (most recent call last):

File "F:/Python/Test/Parent.py", line 38, in <module>

c = Deer()

TypeError: Can't instantiate abstract class Deer with abstract methods sound

**NOTE:** Had there been more than one abstract method in the base class, all of them are required to be implemented in the child classes, otherwise, an error is produced.