# Python Objects and Classes

## Python Classes

A class is considered as a blueprint of objects. We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

Since many houses can be made from the same description, we can create many objects from a class.

An object is simply a collection of data (variables) and methods (functions). Similarly, a class is a blueprint for that object.

## Define Python Class

We use the class keyword to create a class in Python. For example

class ClassName:

# class definition

Here, we have created a class named ClassName.

Let's see an example,

class Bike:

name = ""

gears = 0

price = 0

Here,

* Bike - the name of the class
* name/gear/price - variables inside the class with default values "" and **0** respectively.

**Note**: The variables inside a class are called attributes (Data members).

## Python Objects

An object is called an instance of a class. For example, suppose Bike is a class then we can create objects like bike1, bike2, etc from the class.

Here's the syntax to create an object.

objectName = ClassName()

Let's see an example,

# create class

class Bike:

name = ""

gear = 0

price = 0

# create objects of class

bike1 = Bike()

Here, bike1  is the object of the class. Now, we can use this object to access the class attributes.

## Access Class Attributes Using Objects

We use the . notation to access the attributes of a class. For example,

# modify the name attribute

bike1.name = "Mountain Bike"

# access the gear attribute

bike1.gear

Here, we have used bike1.name and bike1.gear to change and access the value of name and gear attribute respectively.

## Example 1: Python Class and Objects

# define a class

class Bike:

name = ""

gear = 0

# create object of class

bike1 = Bike()

# access attributes and assign new values

bike1.gear = 11

bike1.name = "Mountain Bike"

print(f"Name: {bike1.name}, Gears: {bike1.gear} ")

Run Co

**Output**

Name: Mountain Bike, Gears: 11

In the above example, we have defined the class named Bike with two attributes: name and gear.

We have also created an object bike1 of the class Bike.

Finally, we have accessed and modified the attributes of an object using the . notation.

## Create Multiple Objects of Python Class

We can also create multiple objects from a single class. For example,

# define a class

class Employee:

employee\_id = 0 # define an attribute

employee1 = Employee() # create two objects of the Employee class

employee2 = Employee()

# access attributes using employee1

employee1.employeeID = 1001

print(f"Employee ID: {employee1.employeeID}")

# access attributes using employee2

employee2.employeeID = 1002

print(f"Employee ID: {employee2.employeeID}")

**Output**

Employee ID: 1001

Employee ID: 1002

In the above example, we have created two objects employee1 and employee2 of the Employee class.

## Python Methods and Constructor

We can also define a function inside a Python class. A [Python Function](https://www.programiz.com/python-programming/function) defined inside a class is called a method.

## Python Constructors: we can also initialize data members values using the constructors in python.

## Here, \_\_init\_\_() is the constructor function that is called whenever a new object of that class is instantiated.

# create a class

class Room:

def \_init\_(self,l,b):

self.length = l

self.breadth = b

# method to calculate area

def calculate\_area(self):

print("Area of Room =", self.length \* self.breadth)

# create object of Room class

study\_room = Room(42.5,30.8)

# access method inside class

study\_room.calculate\_area()

Python Classes/Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword class:

Example[Get your own Python Server](https://www.w3schools.com/spaces/)

Create a class named MyClass, with a property named x:

class MyClass:  
  x = 5

Create Object

Now we can use the class named MyClass to create objects:

Example

Create an object named p1, and print the value of x:

p1 = MyClass()  
print(p1.x)

Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

Example

Insert a function that prints a greeting, and execute it on the p1 object:

class Person:  
  def \_\_init\_\_(self, name, age):  
    self.name = name  
    self.age = age  
  def myfunc(self):  
    print("Hello my name is " + self.name)

print("Hello my age is " + self.age)  
  
p1 = Person("John", 36)  
p1.myfunc()

p2=Person(“Amar”, 45)

p2.myfunc()

**Note:** The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

## The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class:

### Example

Use the words mysillyobject and abc instead of self:

class Person:  
  def \_\_init\_\_(mysillyobject, name, age):  
    mysillyobject.name = name  
    mysillyobject.age = age  
  
  def myfunc(abc):  
    print("Hello my name is " + abc.name)  
  
p1 = Person("John", 36)  
p1.myfunc()

## Modify Object Properties

You can modify properties on objects like this:

### Example

Set the age of p1 to 40:

p1.age = 40

class Person:

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

self.name = name

self.age = age

def myfunc(self):

print("Hello my name is " + self.name)

p1 = Person("John", 36)

p1.age = 40

print(p1.age)

## Delete Object Properties

You can delete properties on objects by using the del keyword:

### Example

Delete the age property from the p1 object:

del p1.age

class Person:

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

self.name = name

self.age = age

def myfunc(self):

print("Hello my name is " + self.name)

p1 = Person("John", 36)

del p1.age

print(p1.age)

## Delete Objects

You can delete objects by using the del keyword:

### Example

Delete the p1 object:

del p1

## Delete Objects

You can delete objects by using the del keyword:

### Example

Delete the p1 object:

del p1

del p1.age

class Person:

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

self.name = name

self.age = age

def myfunc(self):

print("Hello my name is " + self.name)

p1 = Person("John", 36)

del p1

print(p1) # it will show error

## The pass Statement

class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the pass statement to avoid getting an error.

### Example

class Person:  
  pass

## Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

**Parent class** is the class being inherited from, also called base class.

**Child class** is the class that inherits from another class, also called derived class.

## Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

### Example

Create a class named Person, with firstname and lastname properties, and a printname method:

class Person:  
  def \_\_init\_\_(self, fname, lname):  
    self.firstname = fname  
    self.lastname = lname  
  def printname(self):  
    print(self.firstname, self.lastname)  
#Use the Person class to create an object, and then execute the printname method:  
x = Person("John", "Doe")  
x.printname()

## Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

### Example

Create a class named Student, which will inherit the properties and methods from the Person class:

class Student(Person):  
  pass

**Note:** Use the pass keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

### Example

Use the Student class to create an object, and then execute the printname method:

x = Student("Mike", "Olsen")  
x.printname()

# Python Inheritance

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.

Python Inheritance

Syntax: **class** derived-**class**(base **class**):

    <**class**-suite>

A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

Example 1

**class** Animal:

**def** speak(self):

**print**("Animal Speaking")  #child class Dog inherits the base class Animal

**class** Dog(Animal):

**def** bark(self):

**print**("dog barking")

d = Dog()

d.bark()

d.speak()

1. **Output:**
2. dog barking
3. Animal Speaking

Python Multi-Level inheritance

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

Python Inheritance

The syntax of multi-level inheritance is given below.

Syntax

**class** class1:

    <**class**-suite>

**class** class2(class1):

    <**class** suite>

**class** class3(class2):

   <**class** suite>

Example

**class** Animal:

**def** speak(self):

**print**("Animal Speaking")

#The child class Dog inherits the base class Animal

**class** Dog(Animal):

**def** bark(self):

**print**("dog barking")

#The child class Dogchild inherits another child class Dog

**class** DogChild(Dog):

**def** eat(self):

**print**("Eating bread...")

d = DogChild()

d.bark()

d.speak()

d.eat()

**Output:**

dog barking

Animal Speaking

Eating bread...

**Example 4:**

#example\_of\_multilevel\_inheritance

class Brands: #parent\_class

brand\_name\_1 = "Amazon"

brand\_name\_2 = "Ebay"

brand\_name\_3 = "OLX"

class Products(Brands): #child\_class

prod\_1 = "Online Ecommerce Store"

prod\_2 = "Online Store"

prod\_3 = "Online Buy Sell Store"

class Popularity(Products): #grand\_child\_class

prod\_1\_popularity = 100

prod\_2\_popularity = 70

prod\_3\_popularity = 60

obj\_1 = Popularity() #Object\_creation

print(obj\_1.brand\_name\_1+" is an "+obj\_1.prod\_1+" popularity of "+str(obj\_1.prod\_1\_popularity))

print(obj\_1.brand\_name\_2+" is an "+obj\_1.prod\_2+" popularity of "+str(obj\_1.prod\_2\_popularity))

print(obj\_1.brand\_name\_3+" is an "+obj\_1.prod\_3+" popularity of "+str(obj\_1.prod\_3\_popularity))

#Output

#Amazon is an Online Ecommerce Store popularity of 100

#Ebay is an Online Store popularity of 70

#OLX is an Online Buy Sell Store popularity of 60

Python Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class.

Python Inheritance

The syntax to perform multiple inheritance is given below.

Syntax

**class** Base1:

    <**class**-suite>

**class** Base2:

    <**class**-suite>

**class** BaseN:

    <**class**-suite>

**class** Derived(Base1, Base2, ...... BaseN):

    <**class**-suite>

Example

**class** Calculation1:

**def** Summation(self,a,b):

**return** a+b;

**class** Calculation2:

**def** Multiplication(self,a,b):

**return** a\*b;

**class** Derived(Calculation1,Calculation2):

**def** Divide(self,a,b):

**return** a/b;

d = Derived()

**print**(d.Summation(10,20))

**print**(d.Multiplication(10,20))

**print**(d.Divide(10,20))

**Output:**

30

200

0.5

**Example 3:**

#example\_of\_multiple\_inheritance

class Brands: #parent\_class

brand\_name\_1 = "Amazon"

brand\_name\_2 = "Ebay"

brand\_name\_3 = "OLX"

class Products: #child\_class

prod\_1 = "Online Ecommerce Store"

prod\_2 = "Online Store"

prod\_3 = "Online Buy Sell Store"

class Popularity(Brands,Products):

prod\_1\_popularity = 100

prod\_2\_popularity = 70

prod\_3\_popularity = 60

obj\_1 = Popularity() #Object\_creation

print(obj\_1.brand\_name\_1+" is an "+obj\_1.prod\_1))

print(obj\_1.brand\_name\_2+" is an "+obj\_1.prod\_2))

print(obj\_1.brand\_name\_3+" is an "+obj\_1.prod\_3))

## 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.

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()

**Example 5:**

#example

class Brands: #parent\_class

brand\_name\_1 = "Amazon"

brand\_name\_2 = "Ebay"

brand\_name\_3 = "OLX"

class Products(Brands): #child\_class

prod\_1 = "Online Ecommerce Store"

prod\_2 = "Online Store"

prod\_3 = "Online Buy Sell Store"

class Popularity(Brands): #grand\_child\_class

prod\_1\_popularity = 100

prod\_2\_popularity = 70

prod\_3\_popularity = 60

class Value(Brands):

prod\_1\_value = "Excellent Value"

prod\_2\_value = "Better Value"

prod\_3\_value = "Good Value"

obj\_1 = Products() #Object\_creation

obj\_2 = Popularity()

obj\_3 = Value()

print(obj\_1.brand\_name\_1+" is an "+obj\_1.prod\_1)

print(obj\_1.brand\_name\_1+" is an "+obj\_1.prod\_1)

print(obj\_1.brand\_name\_1+" is an "+obj\_1.prod\_1)

**Output:**

#Output

#Amazon is an Online Ecommerce Store

#Ebay is an Online Store

#OLX is an Online Buy Sell Store

## 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.

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.

**Example 6:**

#Example\_Hybrid\_inheritance

class PC:

def fun1(self):

print(“This is PC class”)

class Laptop(PC):

def fun2(self):

print(“This is Laptop class inheriting PC class”)

class Mouse(Laptop):

def fun3(self):

print(“This is Mouse class inheriting Laptop class”)

class Student(Mouse, Laptop):

def fun4(self):

print(“This is Student class inheriting PC and Laptop”)

# Driver’s code

obj = Student()

obj1 = Mouse()

obj.fun4()

obj.fun3()

## Add the \_\_init\_\_() Function

So far we have created a child class that inherits the properties and methods from its parent.

We want to add the \_\_init\_\_() function to the child class (instead of the pass keyword).

**Note:** The \_\_init\_\_() function is called automatically every time the class is being used to create a new object.

### Example

Add the \_\_init\_\_() function to the Student class:

class Student(Person):  
  def \_\_init\_\_(self, fname, lname):  
    #add properties etc.

When you add the \_\_init\_\_() function, the child class will no longer inherit the parent's \_\_init\_\_() function.

**Note:** The child's \_\_init\_\_() function **overrides** the inheritance of the parent's \_\_init\_\_() function.

## Use the super() Function

Python also has a super() function that will make the child class inherit all the methods and properties from its parent:

### Example

class Student(Person):  
  def \_\_init\_\_(self, fname, lname):  
    super().\_\_init\_\_(fname, lname)

E.G.

class Person:

def \_\_init\_\_(self, fname, lname):

self.firstname = fname

self.lastname = lname

def printname(self):

print(self.firstname, self.lastname)

class Student(Person):

def \_\_init\_\_(self, fname, lname, year):

super().\_\_init\_\_(fname, lname)

self.graduationyear = year

x = Student("Mike", "Olsen", 2019)

x.printname()

print(x.graduationyear)

## Add Methods

### Example

Add a method called welcome to the Student class:

class Person:

def \_\_init\_\_(self, fname, lname):

self.firstname = fname

self.lastname = lname

def printname(self):

print(self.firstname, self.lastname)

class Student(Person):

def \_\_init\_\_(self, fname, lname, year):

super().\_\_init\_\_(fname, lname)

self.graduationyear = year

def welcome(self):

print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)

x = Student("Mike", "Olsen")

x.printname()

x.welcome()

# Python Polymorphism

The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

## Function Polymorphism

An example of a Python function that can be used on different objects is the len() function.

### String

For strings len() returns the number of characters:

x = "Hello World!"  
  
print(len(x))

### Tuple

For tuples len() returns the number of items in the tuple:

### Example

mytuple = ("apple", "banana", "cherry")  
  
print(len(mytuple))

### Dictionary

For dictionaries len() returns the number of key/value pairs in the dictionary:

### Example

thisdict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}

print(len(thisdict))

## Class Polymorphism

Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.

For example, say we have three classes: Car, Boat, and Plane, and they all have a method called move():

### Example

Different classes with the same method:

class Car:  
  def \_\_init\_\_(self, brand, model):  
    self.brand = brand  
    self.model = model  
  
  def move(self):  
    print("Drive!")  
  
class Boat:  
  def \_\_init\_\_(self, brand, model):  
    self.brand = brand  
    self.model = model  
  
  def move(self):  
    print("Sail!")  
  
class Plane:  
  def \_\_init\_\_(self, brand, model):  
    self.brand = brand  
    self.model = model  
  
  def move(self):  
    print("Fly!")  
  
car1 = Car("Ford", "Mustang")       #Create a Car class  
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat class  
plane1 = Plane("Boeing", "747")     #Create a Plane class  
car1.move()

boat1.move()

plane1.move()

OR  
for x in (car1, boat1, plane1):  
  x.move()

## Inheritance Class Polymorphism

What about classes with child classes with the same name? Can we use polymorphism there?

Yes. If we use the example above and make a parent class called Vehicle, and make Car, Boat, Plane child classes of Vehicle, the child classes inherits the Vehicle methods, but can override them:

### Example

Create a class called Vehicle and make Car, Boat, Plane child classes of Vehicle:

class Vehicle:  
  def \_\_init\_\_(self, brand, model):  
    self.brand = brand  
    self.model = model  
  
  def move(self):  
    print("Move!")  
  
class Car(Vehicle):  
  pass  
  
class Boat(Vehicle):  
  def move(self):  
    print("Sail!")  
  
class Plane(Vehicle):  
  def move(self):  
    print("Fly!")  
  
car1 = Car("Ford", "Mustang") #Create a Car object  
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object  
plane1 = Plane("Boeing", "747") #Create a Plane object  
  
for x in (car1, boat1, plane1):  
  print(x.brand)  
  print(x.model)  
  x.move()

Child classes inherits the properties and methods from the parent class.

In the example above you can see that the Car class is empty, but it inherits brand, model, and move() from Vehicle.

The Boat and Plane classes also inherit brand, model, and move() from Vehicle, but they both override the move() method.

Because of polymorphism we can execute the same method for all classes.

# What is Operator Overloading in Python

The operator overloading in Python means provide extended meaning beyond their predefined operational meaning. Such as, we use the "+" operator for adding two integers as well as joining two strings or merging two lists. We can achieve this as the "+" operator is overloaded by the "int" class and "str" class. The user can notice that the same inbuilt operator or function is showing different behaviour for objects of different classes. This process is known as operator overloading.

**Example:**

1. print (14 + 32)
3. # Now, we will concatenate the two strings
4. print ("Java" + "Tpoint")
6. # We will check the product of two numbers
7. print (23 \* 14)
9. # Here, we will **try** to repeat the String
10. print ("X Y Z " \* 3)

**Output:**

46

JavaTpoint

322

X Y Z X Y Z X Y Z

## How to Overload the Operators in Python?

Suppose the user has two objects which are the physical representation of a user-defined data type class. The user has to add two objects using the "+" operator, and it gives an error. This is because the compiler does not know how to add two objects. So, the user has to define the function for using the operator, and that process is known as "operator overloading". The user can overload all the existing operators by they cannot create any new operator. Python provides some special functions, or we can say magic functions for performing operator overloading, which is automatically invoked when it is associated with that operator. Such as, when the user uses the "+" operator, the magic function \_\_add\_\_ will automatically invoke in the command where the "+" operator will be defined.

## How to Perform Binary "+" Operator in Python:

When the user uses the operator on the user-defined data types of class, then a magic function that is associated with the operator will be invoked automatically. The process of changing the behaviour of the operator is as simple as the behaviour of the function or method defined.

The user define methods or functions in the class and the operator works according to that behaviour defined in the functions. When the user uses the "+" operator, it will change the code of a magic function, and the user has an extra meaning of the "+" operator.

### Program 1: Simply adding two objects.

Python program for simply using the overloading operator for adding two objects.

**Example:**

**class** example:

    def \_\_init\_\_(self, t):

       self.X = t

# adding two objects

    def \_\_add\_\_(self, U):  # here U is considered as another object of

**return** self.X + U.X

obj1 = example( **int**( input( print ("Please enter the value: "))))

obj2 = example( **int**( input( print ("Please enter the value: "))))

print (obj1 + obj2)

ob3 = example(str( input( print ("Please enter the value: "))))

obj4 = example(str( input( print ("Please enter the value: "))))

print (obj3 + obj4)

**Output:**

Please enter the value: 23

Please enter the value: 21

: 44

Please enter the value: Java

Please enter the value: Tpoint

: JavaTpoint

Python program for defining the overloading operator inside another object.

**Example:**

**class** complex\_1:

    def \_\_init\_\_(self, X, Y):

        self.X = X

        self.Y = Y

 # Now, we will add the two objects

    def \_\_add\_\_(self, U):

**return** self.X + U.X, self.Y + U.Y

Object\_1 = complex\_1(23, 12)

Object\_2 = complex\_1(21, 22)

Object\_3 = Object\_1 + Object\_2

print (Object\_3)

**Output:**

(44, 34)

## Python magic functions used for operator overloading:

### Binary Operators:

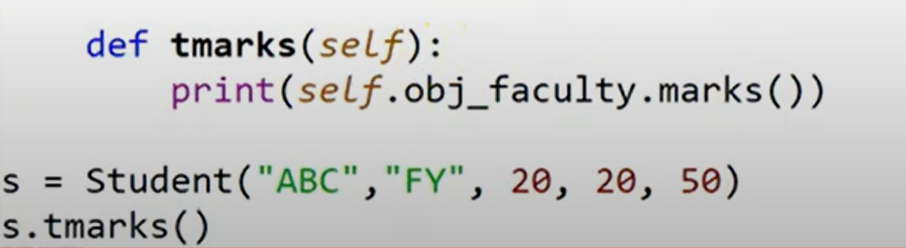
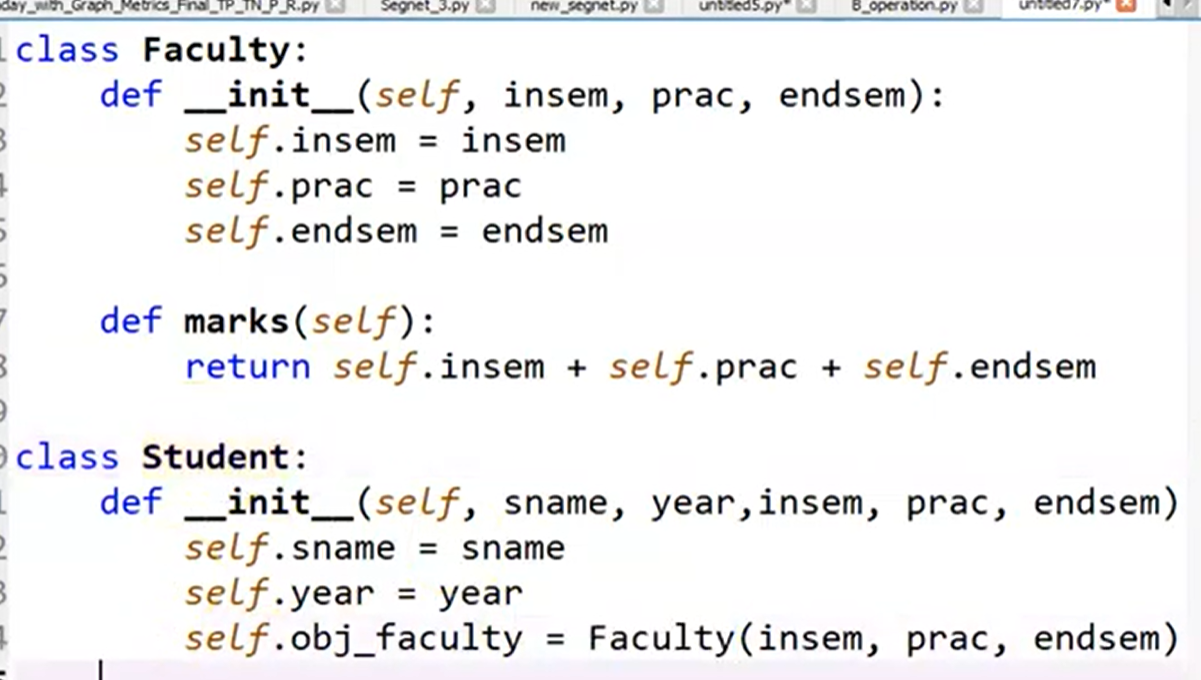
|  |  |
| --- | --- |
| **Operator** | **Magic Function** |
| + | \_\_add\_\_(self, other) |
| - | \_\_sub\_\_(self, other) |
| \* | \_\_mul\_\_(self, other) |
| / | \_\_truediv\_\_(self, other) |
| // | \_\_floordiv\_\_(self, other) |
| % | \_\_mod\_\_(self, other) |
| \*\* | \_\_pow\_\_(self, other) |
| >> | \_\_rshift\_\_(self, other) |
| << | \_\_lshift\_\_(self, other) |
| & | \_\_and\_\_(self, other) |
| | | \_\_or\_\_(self, other) |
| ^ | \_\_xor\_\_(self, other) |

**Delegates and container Class:**

In Python, delegation refers to the process of passing a method call from one object to another object that performs the actual behavior of the method. Delegation is a design pattern that allows objects to collaborate with each other in a flexible and dynamic way.

Delegation can be implemented in Python by defining a class that contains a reference to another object and delegates method calls to that object. The delegated object is typically passed as an argument to the constructor of the delegating class.

When we



Here Student class is a container class and Faculty class is a contained class

# What is a metaclass in Python?

## Metaclasses in Python

Metaclasses are an OOP concept present in all python code by default. Python provides the functionality to create custom metaclasses by using the keyword **type**. Type is a metaclass whose instances are classes. Any class created in python is an instance of type metaclass.

The **type()** function can create classes dynamically as calling type() creates a new instance of type metaclass.

### Syntax

Syntax to create a class using type() is given below −

class name = type(⁢<name>, ⁢<base>, ⁢<dct>)

Where,

<⁢name> is the class name

<⁢bases> defines a tuple of bases class from which the class will inherit

<⁢dct> defines namespace dictionary which contains definition of class

### Example 1

This is the simplest class definition as both and arguments are empty, hence no inheritance and namespace dictionary is null.

DemoClass = type('DemoClass', (), {})

obj = DemoClass()

print(obj)

### Output

The output of the above code gives a variable DemoClass which holds the reference to the class.

<\_\_main\_\_.DemoClass object at 0x7fde96524240>

### Example 2

In the following example, the second parameter is a tuple with a single element Demo from which class Demo2 inherits.

class Demo:

pass

Demo2 = type('Demo2', (Demo,), dict(attribute=10))

obj = Demo2()

print(obj.attribute)

print(obj.\_\_class\_\_)

print(obj.\_\_class\_\_.\_\_bases\_\_)

### Output

In the output of the above program, printing the attribute gives 10 since it has been stored in parameter. Printing the class gives Demo2 as output as that is the derived class and printing the base class gives Demo as the result since it is present in the parameter.

10

⁢class '\_\_main\_\_.Demo2'>

(⁢class '\_\_main\_\_.Demo'>,)

### Example 3

In the following example the <dct>  parameter has an attribute variable and attribute\_value which acts as the method of Demo class.

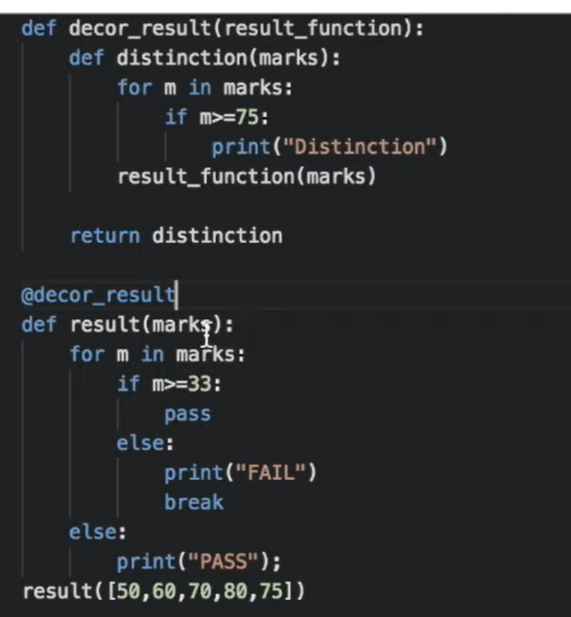
**Decorators in Python**

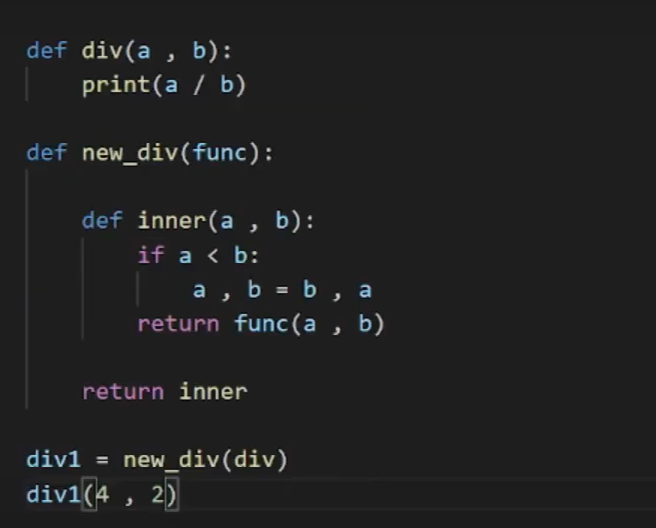
In Python, a decorator is a design pattern that allows you to modify the functionality of a function by wrapping it in another function.

The outer function is called the decorator, which takes the original function as an argument and returns a modified version of it.

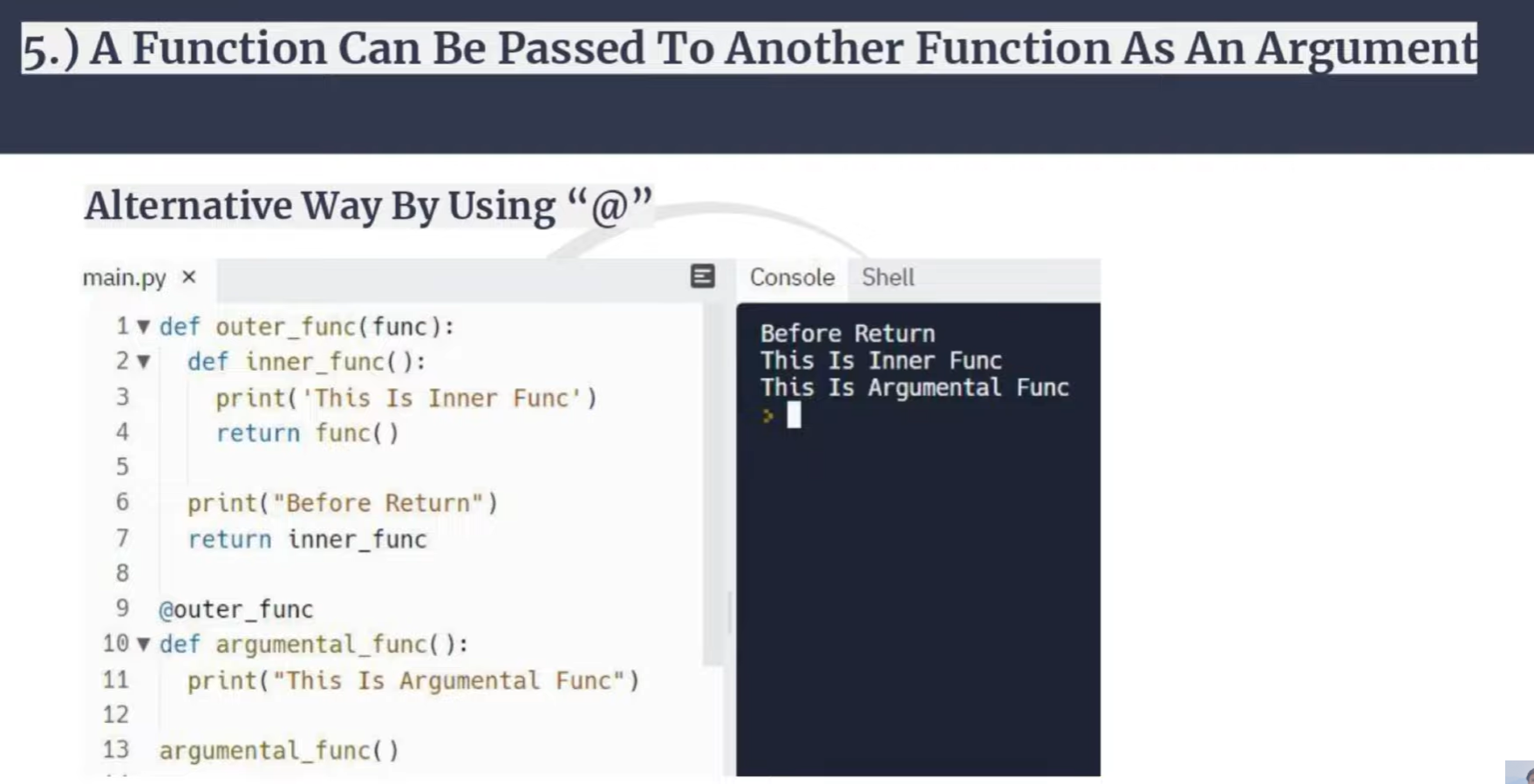
A Python decorator is a function that takes in a function and returns it by adding some functionality. Basically, a decorator takes in a function as an argument then adds some functionality into it and returns it. When we need a function that change its behaviour in some cases but it will be same in all remaining cases. Whenever we use this decorator in that case only the original function’s behaviour will be changed.

e.g.

****

****

**e.g.**

****

**Static Method**

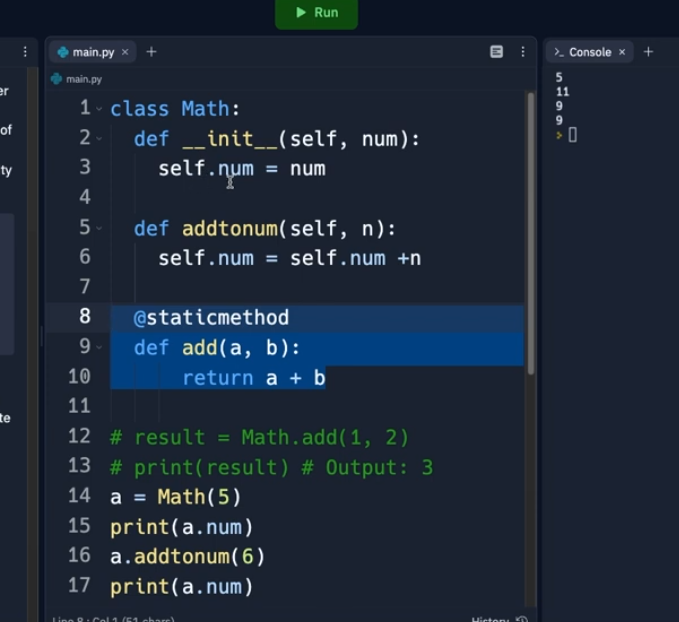
We know that a static method in Python is treated differently than in other languages. Static methods in Python are extremely similar to python class methods, but the difference is that a static method is bound to a class rather than the objects for that class.  
To define a static method, we use the **@staticmethod** decorator, which is a built-in decorator. Also, there is no need to import any module to use decorators. Using a static method in a class, we permit it to be accessed only by the class objects or inside the class.

**There are few limitations related to static methods, such as:**

* Unlike, class method, a static method cannot alter or change any variable value or state of the class.
* Static methods do not have any knowledge related to the class.
* **In Python, static methods can be created using @staticmethod.**
* class Student:
* @staticmethod
* def myfunc():
* //Code to be executed

#### Advantages of Python static method

Static methods have a very clear use case. When we need some functionality not for an Object but with the complete class, we make a method static. This is advantageous when we need to create utility methods.  
**Finally, note that we do not need the self or cls to be passed as the first argument in a static method**.



## Introduction to the Python Metaclass

A metaclass is a [class](https://www.pythontutorial.net/python-oop/python-class/) that creates other classes. By default, Python uses the [type](https://www.pythontutorial.net/python-oop/python-type-class/) metaclass to create other classes.

For example, the following defines a Person class:

class Person:

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

self.name = name

self.age = age

When Python executes the code, it uses the type metaclass to create the Person class. The reason is that the Person class uses the type metaclass by default.

The explicit Person class definition looks like this:

class Person(object, metaclass=type):

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

self.name = name

self.age = age

The metaclass argument allows you to specify which metaclass id used to define the class. Therefore, you can create a custom metaclass that has its own logic to create other classes. By using a custom metaclass, you can inject functionality into the class creation process.

## Python metaclass example

First, define a custom metaclass called Human that has the freedom attribute sets to True by default:

class Human(type):

def \_\_new\_\_(mcs, name, bases, class\_dict):

class\_ = super().\_\_new\_\_(mcs, name, bases, class\_dict)

class\_.freedom = True

return class\_

Note that the [\_\_new\_\_](https://www.pythontutorial.net/python-oop/python-__new__/) method returns a new class or a class object.

Second, define the Person class that uses the Human metaclass:

class Person(object, metaclass=Human):

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

self.name = name

self.age = age

The Person class will have the freedom attribute as shown in the [class variables](https://www.pythontutorial.net/python-oop/python-class-variables/):

pprint(Person.\_\_dict\_\_)

Output:

mappingproxy({'\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Person' objects>,

'\_\_doc\_\_': None,

'\_\_init\_\_': <function Person.\_\_init\_\_ at 0x000001E716C71670>,

'\_\_module\_\_': '\_\_main\_\_',

'\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Person' objects>,

'freedom': True})Code language: Python (python)

Put it all together.

from pprint import pprint

class Human(type):

def \_\_new\_\_(mcs, name, bases, class\_dict):

class\_ = super().\_\_new\_\_(mcs, name, bases, class\_dict)

class\_.freedom = True

return class\_

class Person(object, metaclass=Human):

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

self.name = name

self.age = age

pprint(Person.\_\_dict\_\_)Code language: Python (python)

## Metaclass Parameters

To pass parameters to a metaclass, you use the keyword arguments. For example, the following redefines the Human metaclass that accepts keyword arguments, where each argument becomes a class variable:

class Human(type):

def \_\_new\_\_(mcs, name, bases, class\_dict, \*\*kwargs):

class\_ = super().\_\_new\_\_(mcs, name, bases, class\_dict)

if kwargs:

for name, value in kwargs.items():

setattr(class\_, name, value)

return class\_Code language: Python (python)

The following uses the Human metaclass to create a Person class with the country and freedom class variables set to USA and True respectively:

class Person(object, metaclass=Human, country='USA', freedom=True):

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

self.name = name

self.age = ageCode language: Python (python)

Here are Person class variables:

pprint(Person.\_\_dict\_\_)Code language: Python (python)

Output:

mappingproxy({'\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Person' objects>,

'\_\_doc\_\_': None,

'\_\_init\_\_': <function Person.\_\_init\_\_ at 0x0000018A334235E0>,

'\_\_module\_\_': '\_\_main\_\_',

'\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Person' objects>,

'country': 'USA',

'freedom': True})Code language: Python (python)

Put it all together.

from pprint import pprint

class Human(type):

def \_\_new\_\_(mcs, name, bases, class\_dict, \*\*kwargs):

class\_ = super().\_\_new\_\_(mcs, name, bases, class\_dict)

if kwargs:

for name, value in kwargs.items():

setattr(class\_, name, value)

return class\_

class Person(object, metaclass=Human, freedom=True, country='USA'):

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

self.name = name

self.age = age

pprint(Person.\_\_dict\_\_)Code language: Python (python)

## When to use metaclasses

Here’s the quote from [Tim Peter](https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)) who wrote the Zen of Python:

Metaclasses are deeper magic that 99% of users should never worry about it. If you wonder whether you need them, you don’t (the people who actually need them to know with certainty that they need them and don’t need an explanation about why).

Tim Peter

In practice, you often don’t need to use metaclasses unless you maintain or develop the core of large frameworks such as Django.

## Summary

* A metaclass is a class that creates other classes.

## What is Namespace?

In Python, a way to give each object a unique name is through a namespace. Variables and methods are examples of objects in Python. To put it another way, it is a collection of the known symbolic names and the details about the thing that each name refers to. A name can be thought of as a key in a dictionary, and objects are the values in a namespace. We should figure out it with a genuine model - A namespace resembles a last name. If there are multiple "Peter" names in the class, it may be difficult to locate a "Peter" name; however, when we specifically request "Peter Warner" or "Peter Cummins," In a class, it might not be common for multiple students to have the same first and last name. In Python, there are four types of namespaces which are given below.

* Built-in
* Global
* Enclosing
* Local

The Built-in Namespace

As its name suggests, it contains pre-defined names of all of Python's built-in objects already available in Python. Let's list these names with the following command.

Open the Python terminal and type the following command.

**Command -**

1. dir(\_\_builtins\_\_)

## The Global Namespace:

## The global namespace consists of any names in Python at any level of the main program. It is created when the main body executes and remains in existence until the interpreter terminates.

The Python interpreter creates a global namespace for any module that our Python loads with the import statement. To get more information, visit our Python Module