

# Python-Classes & Inheritance

## Classes and Objects

A class is a user-defined blueprint that encapsulates related data and behavior into a single entity. It serves as a template for creating objects, which are instances of that class. With a single class definition, you can instantiate multiple objects, each inheriting the properties and methods outlined within the class.

python classes are composed of two types of attributes:

### Class Attributes:

- **Shared by all instances:** Class attributes belong to the class itself. A change in a class attribute affects all instances of that class.
- **Defined outside of methods:** They are declared directly within the class body but outside of any methods (including the `__init__` method).

### Instance Attributes:

- **Specific to each object:** Instance attributes are unique to each object (or instance) of a class. Their values can vary between different objects of the same class.
- **Defined in the `__init__()` method:** Instance attributes are typically initialized within the constructor (`__init__`) of the class.
- **Represent object state:** Instance attributes store the data that defines the particular characteristics or state of a specific object.

### Syntax:

```
class ClassName:
    """
    This is a docstring.
    It provides a brief description of the class.
    """
    # Class attributes and methods go here
    pass
```

In this structure:

- **class ClassName:** defines a new class named ClassName.
- **Docstring:** The triple-quoted string immediately following the class definition is called a docstring. While not mandatory, it is a recommended practice to include a docstring that describes the purpose and functionality of the class, improving code readability and maintainability.

# Python-Classes & Inheritance

- **Statements:** Within the class definition, you can define class attributes (variables) and methods (functions).
- **Pass:** It is used to indicate that this class is empty.

## `__init__()` method:

The `__init__` method is a special method in Python classes, which is automatically called when an object of the class is created. It is similar to constructors in other object-oriented programming languages like Java and C++. The `__init__` method is used to initialize the attributes of an object.

Here's an example:

```
class Person:
    def __init__(self, name, age):
        self.name = name # Assigning the name attribute
        self.age = age # Assigning the age attribute

# Creating an object
person1 = Person("Alice", 25)
print(person1.name) # Output: Alice
print(person1.age) # Output: 25
```

## Self Parameter:

In Python, **self** is a reference to the current instance of a class. It is used to access and modify the attributes and methods of the class within the class itself. While **self** is a widely accepted convention, you can use any other name, but it's recommended to stick with **self** for better readability and consistency with Python's conventions. It's important to note that **self** is not a reserved keyword in Python;

## Syntax:

```
class ClassName:
    def __init__(self, arg1, arg2):
        self.attr1 = arg1 # Assigning attribute using self
        self.attr2 = arg2

    def method_name(self, arg):
        # Access instance attributes using self
        result = self.attr1 + arg
```

# Python-Classes & Inheritance

```
# Modify instance attributes using self
self.attr2 = result
# Return a value
return self.attr2
```

## Constructors in python:

A constructor is a special method used to create and initialize an object of a class. This method is defined within the class itself. It is automatically executed when an object is created, and its primary purpose is to declare and initialize the object's data members or instance variables.

The constructor contains a collection of statements that execute at the time of object creation, setting up the initial state of the object's attributes. For instance, when you execute **obj = Sample()**, Python recognizes that **obj** is an object of the Sample class and calls the constructor of that class to create and initialize the object.

## Syntax:

```
def __init__(self):
    # body of the constructor
```

Where,

- **def:** The keyword is used to define function.
- **\_\_init\_\_ () Method:** It is a reserved method. This method gets called as soon as an object of a class is instantiated.
- **self:** The first argument self refers to the current object. It binds the instance to the **init()** method. It's usually named self to follow the naming convention.

## Types of Constructors:

In Python, there are three main types of constructors:

### 1. Default Constructor:

- This is the constructor that is automatically provided by Python when no constructor is explicitly defined in the class.
- It is an empty constructor without any parameters.
- The default constructor is used to create an object without any initial values for the object's attributes.

### 2. Non-Parameterized Constructor:

# Python-Classes & Inheritance

- This is a constructor that is defined without any parameters.
- It is used to initialize the object's attributes with default or predefined values.
- Although it doesn't take any arguments, it can still perform initialization tasks or call other methods within the class.

### 3. Parameterized Constructor:

- This is a constructor that accepts one or more parameters.
- The parameters are used to initialize the object's attributes with specific values passed during object creation.
- Parameterized constructors allow for more flexible and customized object initialization based on the values provided.

It's important to note that in Python, there is no explicit syntax for defining different types of constructors. The constructor is always defined using the `__init__` method, and the number and types of parameters it accepts determine whether it is a default, non-parameterized, or parameterized constructor.

Here's an example illustrating the different types of constructors:

```
class Example:
    # Default Constructor
    def __init__(self):
        self.value = 0

    # Non-Parameterized Constructor
    def __init__(self):
        self.value = 10

    # Parameterized Constructor
    def __init__(self, value):
        self.value = value
```

## What is an Object?

An object is a concrete realization of a class, an instantiation of the blueprint defined by the class. It encapsulates a collection of data (attributes or variables) and associated operations (methods) that operate on that data. Objects are used to perform actions and represent real-world entities or concepts within a program.

## Syntax:

# Python-Classes & Inheritance

```
object_name = ClassName()
```

## Attributes:

Attributes are the properties or characteristics that define the state or qualities of an object. They differentiate one object from another within the same class. Attributes are declared as variables within a class, and each object can have its own unique values for these variables.

There are two types of attributes:

### Class Attribute:

These attributes are shared among all instances of the class. There is only one copy of a class attribute, and any modifications made to it will be reflected across all instances.

```
class Player:
    team_name = "Dragons" # Class attribute

    def __init__(self, name):
        self.name = name

player1 = Player("Alice")
player2 = Player("Bob")

print(player1.team_name) # Output: Dragons
print(player2.team_name) # Output: Dragons
```

### Instance Attribute:

These attributes are specific to each instance (object) of the class. Every object has its own copy of instance attributes, and any changes made to these attributes affect only that particular object.

```
class Player:
    # ... (previous code) ...

    def __init__(self, name, score):
        self.name = name
        self.score = score

player1 = Player("Alice", 100)
```

# Python-Classes & Inheritance

```
player2 = Player("Bob", 50)
```

```
print(player1.score) # Output: 100
```

```
print(player2.score) # Output: 50
```

main.py	Save	Run	Output
<pre>1 class Person: 2     def __init__(self, name, age): 3         self.name = name 4         self.age = age 5 6 p1 = Person("John", 36) 7 8 print(p1.name) 9 print(p1.age) 10</pre>			<pre>John 36  === Code Execution Successful ===</pre>

#code:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
p1 = Person("John", 36)
```

```
print(p1.name)
```

```
print(p1.age)
```

## Object Method:

main.py	Save	Run	Output
<pre>1 class Person: 2     def __init__(self, name, age): 3         self.name = name 4         self.age = age 5 6     def myfunc(self): 7         print("Hello my name is " + self.name) 8 9 p1 = Person("John", 36) 10 p1.myfunc() 11</pre>			<pre>Hello my name is John  === Code Execution Successful ===</pre>

#code:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

# Python-Classes & Inheritance

```
def myfunc(self):  
    print("Hello my name is " + self.name)
```

```
p1 = Person("John", 36)  
p1.myfunc()
```

## Self Parameters:

main.py	Save	Run	Output
<pre>1 class Person: 2     def __init__(mysillyobject, name, age): 3         mysillyobject.name = name 4         mysillyobject.age = age 5 6     def myfunc(abc): 7         print("Hello my name is " + abc.name) 8 9 p1 = Person("John", 36) 10 p1.myfunc() 11</pre>			<pre>Hello my name is John  === Code Execution Successful ===</pre>

## Modify Objects:

main.py	Save	Run	Output
<pre>1 class Person: 2     def __init__(self, name, age): 3         self.name = name 4         self.age = age 5 6     def myfunc(self): 7         print("Hello my name is " + self.name) 8 9 p1 = Person("John", 36) 10 11 p1.age = 40 12 13 print(p1.age)</pre>			<pre>40  === Code Execution Successful ===</pre>

class Person:

```
def __init__(self, name, age):
```

```
    self.name = name
```

```
    self.age = age
```

```
def myfunc(self):
```

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

# Python-Classes & Inheritance

```
p1 = Person("John", 36)
```

```
p1.age = 40
```

```
print(p1.age)
```

## OOP Concepts:

### What is Object-Oriented Programming in Python?

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 programming. The main concept of object-oriented Programming (OOPs) or oops concepts in Python is to bind the data and the functions that work together as a single unit so that no other part of the code can access this data.

### OOPs Concepts in Python

- Polymorphism in Python
- Encapsulation in Python
- Inheritance in Python
- Data Abstraction in Python

## Python Inheritance

In Python object oriented Programming, Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class. The benefits of inheritance are:

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

### Types of Inheritance

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



# Python-Classes & Inheritance

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

## Inheritance in Python

In the above article, we have created two classes i.e. Person (parent class) and Employee (Child Class). The Employee class inherits from the Person class. We can use the methods of the person class through the employee class as seen in the display function in the above code. A child class can also modify the behavior of the parent class as seen through the details() method.

Python Inheritance code:

Try it yourself:

# 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)

## Python-Classes & Inheritance

```
print(self.idnumber)
```

```
def details(self):
```

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

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

```
# child class
```

```
class Employee(Person):
```

```
    def __init__(self, name, idnumber, salary, post):
```

```
        self.salary = salary
```

```
        self.post = post
```

```
    # invoking the __init__ of the parent class
```

```
    Person.__init__(self, name, idnumber)
```

```
def details(self):
```

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

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

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

```
# creation of an object variable or an instance
```

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

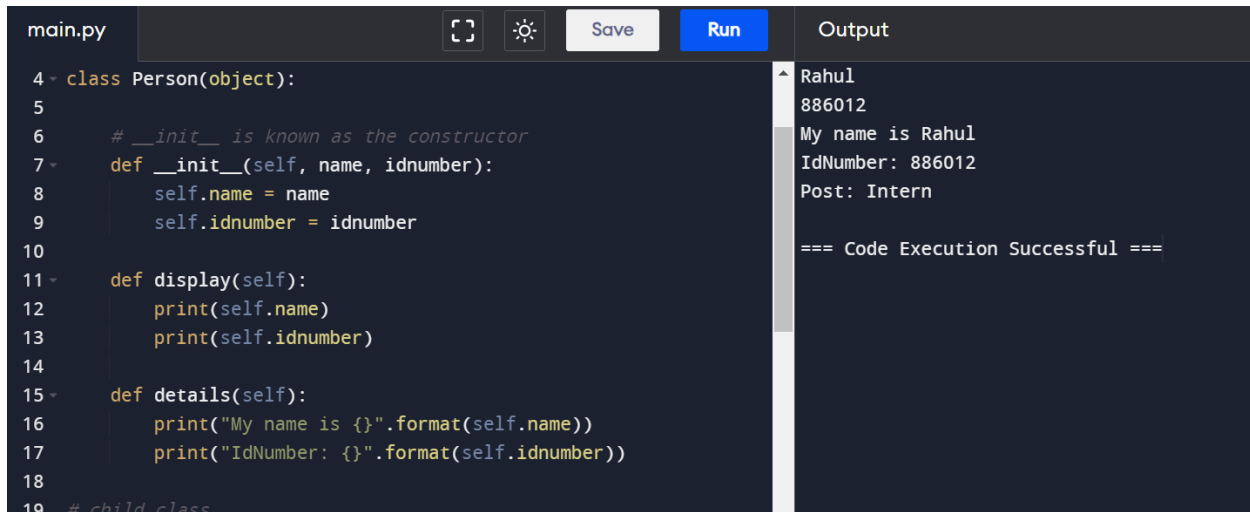
```
# calling a function of the class Person using
```

```
# its instance
```

```
a.display()
```

# Python-Classes & Inheritance

a.details()



The screenshot shows a Python IDE with a file named 'main.py'. The code defines a 'Person' class with an '\_\_init\_\_' method (constructor) that takes 'name' and 'idnumber' as arguments and assigns them to 'self.name' and 'self.idnumber'. It also has a 'display' method that prints the name and idnumber, and a 'details' method that prints a formatted string. The output window on the right shows the results of running the code: 'Rahul', '886012', 'My name is Rahul', 'IdNumber: 886012', 'Post: Intern', and a success message '=== Code Execution Successful ==='.

```
main.py  [ ] [ ] Save Run Output
4 class Person(object):
5
6     # __init__ is known as the constructor
7     def __init__(self, name, idnumber):
8         self.name = name
9         self.idnumber = idnumber
10
11     def display(self):
12         print(self.name)
13         print(self.idnumber)
14
15     def details(self):
16         print("My name is {}".format(self.name))
17         print("IdNumber: {}".format(self.idnumber))
18
19 # child class
```

Output

Rahul  
886012  
My name is Rahul  
IdNumber: 886012  
Post: Intern  
=== Code Execution Successful ===

## Python Polymorphism

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

## Polymorphism in Python

This code demonstrates the concept of Python oops inheritance and method overriding in Python classes. It shows how subclasses can override methods defined in their parent class to provide specific behavior while still inheriting other methods from the parent class.

```
class Bird:
```

```
    def intro(self):
```

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

```
    def flight(self):
```

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

# Python-Classes & Inheritance

```
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()
```

# Python-Classes & Inheritance

```
main.py  [ ] [ ] Save Run Output
1 class Bird:
2
3     def intro(self):
4         print("There are many types of birds.")
5
6     def flight(self):
7         print("Most of the birds can fly but some cannot.")
8
9 class sparrow(Bird):
10
11     def flight(self):
12         print("Sparrows can fly.")
13
14 class ostrich(Bird):
15
16     def flight(self):
17         print("Ostriches cannot fly.")
```

Output

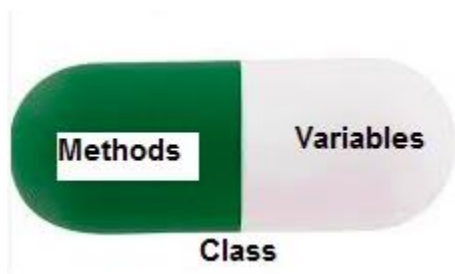
```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.

=== Code Execution Successful ===
```

## Python Encapsulation

In Python object oriented programming, 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.



```
class Student:

    def __init__(self, name="Rajaram", marks=50):

        self.name = name

        self.marks = marks

s1 = Student()
```

# Python-Classes & Inheritance

```
s2 = Student("Bharat", 25)
```

```
print ("Name: {} marks: {}".format(s1.name, s2.marks))
```

```
print ("Name: {} marks: {}".format(s2.name, s2.marks))
```

main.py	Output
<pre>1 - class Student: 2 -     def __init__(self, name="Rajaram", marks=50): 3         self.name = name 4         self.marks = marks 5 6 s1 = Student() 7 s2 = Student("Bharat", 25) 8 9 print ("Name: {} marks: {}".format(s1.name, s2.marks)) 10 print ("Name: {} marks: {}".format(s2.name, s2.marks))</pre>	<pre>Name: Rajaram marks: 25 Name: Bharat marks: 25  === Code Execution Successful ===</pre>

## Abstraction classes in Python

In Python, abstraction can be achieved by using abstract classes and interfaces.

A class that consists of one or more abstract method is called the abstract class. Abstract methods do not contain their implementation. Abstract class can be inherited by the subclass and abstract method gets its definition in the subclass. Abstraction classes are meant to be the blueprint of the other class. An abstract class can be useful when we are designing large functions. An abstract class is also helpful to provide the standard interface for different implementations of components. Python provides the **abc** module to use the abstraction in the Python program. Let's see the following syntax.

### Try yourself:

```
# Python program demonstrate
# abstract base class work
from abc import ABC, abstractmethod
class Car(ABC):
    def mileage(self):
        pass
```

# Python-Classes & Inheritance

```
class Tesla(Car):
    def mileage(self):
        print("The mileage is 30kmph")
class Suzuki(Car):
    def mileage(self):
        print("The mileage is 25kmph ")
class Duster(Car):
    def mileage(self):
        print("The mileage is 24kmph ")

class Renault(Car):
    def mileage(self):
        print("The mileage is 27kmph ")

# Driver code
t= Tesla ()
t.mileage()

r = Renault()
r.mileage()

s = Suzuki()
s.mileage()
d = Duster()
d.mileage()
```

# Python-Classes & Inheritance

```
main.py  [ ] [ ] Save Run Output
1 # Python program demonstrate
2 # abstract base class work
3 from abc import ABC, abstractmethod
4 class Car(ABC):
5     def mileage(self):
6         pass
7
8 class Tesla(Car):
9     def mileage(self):
10        print("The mileage is 30kmph")
11 class Suzuki(Car):
12     def mileage(self):
13        print("The mileage is 25kmph ")
14 class Duster(Car):
15     def mileage(self):
16        print("The mileage is 24kmph ")
17
18 class Renault(Car):
19     def mileage(self):
20        print("The mileage is 27kmph ")

The mileage is 30kmph
The mileage is 27kmph
The mileage is 25kmph
The mileage is 24kmph

=== Code Execution Successful ===
```

## Inheritance:

Inheritance is a key concept in object-oriented programming (OOP) that allows a class (subclass or derived class) to inherit properties and methods from another class (superclass or base class).

Creating a Base class:

**class Animal:**

```
def __init__(self, name):
    self.name = name
```

```
def speak(self):
```

```
    raise NotImplementedError("Subclass must implement this method")
```

```
main.py  [ ] [ ] Save Run Output
1 class Animal:
2     def __init__(self, name):
3         self.name = name
4
5     def speak(self):
6         raise NotImplementedError("Subclass must implement this method")

=== Code Execution Successful ===
```

#code:

**class Animal:**



# Python-Classes & Inheritance

```
def __init__(self, name):
```

```
    self.name = name
```

```
def speak(self):
```

```
    raise NotImplementedError("Subclass must implement this method")
```

**Try it yourself:**

**Exercise1:**

**Creating Derived class:**

```
class Animal:
```

```
    def __init__(self, name):
```

```
        self.name = name
```

```
class Dog(Animal):
```

```
    def speak(self):
```

```
        return f"{self.name} says Woof!"
```

**Exercise 2: Creating Class Hierarchy**

```
class Animal:
```

```
    def __init__(self, name):
```

```
        self.name = name
```

```
    def speak(self):
```

```
        raise NotImplementedError("Subclass must implement this method")
```

```
class Dog(Animal):
```

```
    def speak(self):
```

# Python-Classes & Inheritance

```
return f"{self.name} says Woof!"
```

```
class Bird:
```

```
    def fly(self):
```

```
        return "Flying high"
```

```
class Parrot(Animal, Bird):
```

```
    def speak(self):
```

```
        return f"{self.name} says Squawk!"
```

```
print(issubclass(Dog, Animal)) # Output: True
```

```
print(issubclass(Parrot, Bird)) # Output: True
```

Exercise 3:

```
class Animal:
```

```
    def __init__(self, name):
```

```
        self.name = name
```

```
    def speak(self):
```

```
        raise NotImplementedError("Subclass must implement this method")
```

```
class Dog(Animal):
```

```
    def speak(self):
```

```
        return f"{self.name} says Woof!"
```

```
class Bird:
```

```
    def fly(self):
```

```
        return "Flying high"
```

```
class Parrot(Animal, Bird):
```

```
    def speak(self):
```

```
        return f"{self.name} says Squawk!"
```

```
class Cat(Animal):
```

```
    def speak(self):
```

```
        return f"{self.name} says Meow!"
```

# Python-Classes & Inheritance

```
cat = Cat("Whiskers")  
print(cat.speak()) # Output: "Whiskers says Meow!"
```

# Conclusion:

- The Cat class inherits from Animal and overrides the `speak()` method to provide a customized behavior of printing a meow message.
- The cat object is then created and its speak method is called to confirm the overridden behavior.

## Iterators:

- Iterators are objects in Python that allow for looping through a sequence of elements, such as lists, tuples, or dictionaries.
- They provide a way to access elements one by one without needing to know the underlying data structure.

### Try it yourself:

#### Exercise 1:

```
numbers = [1, 2, 3, 4, 5]  
for num in numbers:  
    print(num)
```

#### Exercise 2:

```
numbers = [1, 2, 3, 4, 5]  
iter_numbers = iter(numbers)  
print(next(iter_numbers)) # Output: 1  
print(next(iter_numbers)) # Output: 2
```

# Notes:

- The `iter()` function creates an iterator object from the numbers list.
- The `next()` function is used to retrieve elements from the iterator sequentially.
- After each call to `next()`, the iterator advances to the next element

# Python-Classes & Inheritance

## Exercise 3:

```
def square_numbers(nums):
    for num in nums:
        yield num * num
numbers = [1, 2, 3, 4, 5]
squares = square_numbers(numbers)
for square in squares:
    print(square)
```

## Try it yourself:

### Built-in Iterators:

#### a. range() function:

```
for i in range(5):
    print(i)
```

main.py	Save	Run	Output
<pre>1 fruits = ['apple', 'banana', 'cherry'] 2 for index, fruit in enumerate(fruits): 3     print(index, fruit) 4</pre>			<pre>0 apple 1 banana 2 cherry  === Code Execution Successful ===</pre>

## Scope:

Scope is the portion of code that a variable or named Python object is accessible from. There are four scopes:

- local - a code block, such as the body of a function or the suite that follows a conditional statement or loop.
- enclosing - if there is a function or code block within a function, then the inner scope can access the variables available in the enclosing function.
- global - the current module. This is the outermost scope an object can be in.
- built-in - the automatically imported module, built-ins.

## Local Scope:

# Python-Classes & Inheritance

main.py	Output
<pre>1 def myfunc(): 2     x = 300 3     print(x) 4 5 myfunc() 6</pre>	<pre>300  === Code Execution Successful ===</pre>

## Enclosing Scope:

main.py	Output
<pre>1 def myfunc(): 2     x = 300 3     def myinnerfunc(): 4         print(x) 5     myinnerfunc() 6 7 myfunc() 8</pre>	<pre>300  === Code Execution Successful ===</pre>

## Global Scope:

main.py	Output
<pre>1 x = 300 2 3 def myfunc(): 4     print(x) 5 6 myfunc() 7 8 print(x)</pre>	<pre>300 300  === Code Execution Successful ===</pre>

*Thank you*