Q1. What is the purpose of Python's OOP?

Ans:- In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims **to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming**.

Q2. Where does an inheritance search look for an attribute?

Ans:- We've been using this expression throughout the book to access module attributes, call methods of objects, and so on. When we say this to an object that is derived from a class statement, however, the expression kicks off a search in Python—it searches a tree of linked objects, looking for the first appearance of attribute that it can find. When classes are involved, the preceding Python expression effectively translates to the following in natural language:

Find the first occurrence of attribute by looking in object, then in all classes above it, from bottom to top and left to right.

In other words, attribute fetches are simply tree searches. The term inheritance is applied because objects lower in a tree inherit attributes attached to objects higher in that tree. As the search proceeds from the bottom up, in a sense, the objects linked into a tree are the union of all the attributes defined in all their tree parents, all the way up the tree.

In Python, this is all very literal: we really do build up trees of linked objects with code, and Python really does climb this tree at runtime searching for attributes every time we use the object. attribute expression. To make this more concrete, Figure 25-1 sketches an example of one of these trees.

In this figure, there is a tree of five objects labeled with variables, all of which have attached attributes, ready to be searched. More specifically, this tree links together three

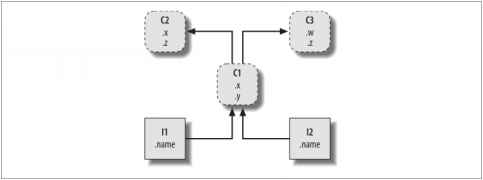


Figure 25-1. A class tree, with two instances at the bottom (I1 and I2), a class above them (C1), and two superclasses at the top (C2 and C3). All of these objects are namespaces (packages of variables), and the inheritance search is simply a search of the tree from bottom to top looking for the lowest occurrence of an attribute name. Code implies the shape of such trees.

Figure 25-1. A class tree, with two instances at the bottom (I1 and I2), a class above them (C1), and two superclasses at the top (C2 and C3). All of these objects are namespaces (packages of variables), and the inheritance search is simply a search of the tree from bottom to top looking for the lowest occurrence of an attribute name. Code implies the shape of such trees.

class objects (the ovals C1, C2, and C3) and two instance objects (the rectangles I1 and I2) into an inheritance search tree. Notice that in the Python object model, classes and the instances you generate from them are two distinct object types

Q3. How do you distinguish between a class object and an instance object?

## Ans:- class object:

when we **create**a class in python then a **class object** is created so whenever python finds a class statement in the whole program then it creates a class object and assigns a name to that object i.e. class name. As we know in python, everything is an object so the class itself is an object and is the instance of [metaclasses](https://artificialintelligencestechnology.com/python/metaclasses-in-python/). Look at the following example

|  |  |
| --- | --- |
| 1  2 | class MyClass:   pass |

above code will generate a class object and name it ‘MyClass’. From this class object, we will create **instance objects**.

Class objects provide default behavior and serve as factories for instance objects.

the class object comes from the ‘class’ statement in code. whenever we encounter a class statement then a **class object** will be created.

class object inherits the attributes of its parent classes.

Instance object:

when we **call**a class, it creates an instance object of that class from which the object has been created. For example when we call the above-created class then it will create an instance object like this.

|  |  |
| --- | --- |
| 1 | Obj1=MyClass() |

the above statement creates an object and names it to **Obj1**which is an instance of MyClass.

Instance objects are real objects in your python code process. The instance object has access to attributes of the class from which it is created. For example, **Obj1**is the instance of class **MyClass** so, Now Obj1 can access everything defined in the class, and in the class object, we define the default behavior and properties of objects.

The instance object comes from a call i.e. when we call the class. Actually, we are creating instance objects of that class.

instance object inherits the attributes of the class object from which it was created.

* class object is like a blueprint for intance object but instance object is a concrete item in out code.
* instance objects are new namespaces, thay start out empty but inherit object attributes that live in class object.
* The first argument of class functions(self) reference the instance object and assignments to attributes of self change data in the instance.

Q4. What makes the first argument in a class’s method function special?

Ans:- **Class method**: Used to access or modify the class state. In method implementation, if we use only [class variables](https://pynative.com/python-class-variables/), then such type of methods we should declare as a class method. The class method has a cls parameter which refers to the class.

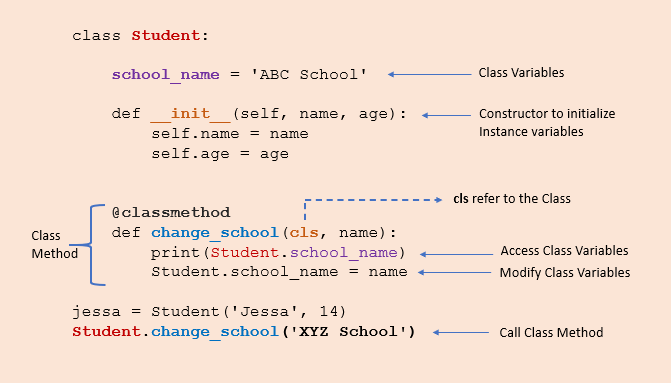
Class methods are methods that are called on the [class](https://pynative.com/python-classes-and-objects/) itself, not on a specific object instance. Therefore, it belongs to a class level, and all class instances share a class method.

* **A class method is bound to the class** and not the object of the class. It can access only class variables.
* It can modify the class state by changing the value of a [class variable](https://pynative.com/python-class-variables/) that would apply across all the class objects.

In method implementation, if we use only class variables, we should declare such methods as class methods. The class method has a cls as the first parameter, which refers to the class.

Class methods are used when we are **dealing with factory methods**. Factory methods are those methods that **return a class object for different use cases**. Thus, factory methods create concrete implementations of a common interface.

The class method can be called using ClassName.method\_name() as well as by using an object of the class.



Q5. What is the purpose of the init method?

Ans:- The \_\_init\_\_ method lets the class initialize the object’s attributes and serves no other purpose. It is only used within classes.

### Create a Class

Let’s begin by creating a class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class Dog:    def \_\_init\_\_(self,dogBreed,dogEyeColor):        self.breed = dogBreed      self.eyeColor = dogEyeColor... |

First, we declare the class Dog using the keyword class. We use the keyword def to define a function or method, such as the \_\_init\_\_ method. As you can see, the \_\_init\_\_ method initializes two attributes: breed and eyeColor.

We’ll now see how to pass these parameters when declaring an object. This is where we need the keyword self to bind the object’s attributes to the arguments received.

Q6. What is the process for creating a class instance?

Ans:- To create instances of a class, you call the class using class name and pass in whatever arguments its \_\_init\_\_ method accepts.

class Employee:

# Class attribute

empCount = 0

# Constructor of class

# it is mainly used for assignment of instance variables

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

# instance variable or instance attributes

self.emp\_name = name

self.emp\_salary = salary

Employee.empCount += 1

# method of a class

def displayEmployeeInfo(self):

print("Employee name : ",self.emp\_name, " , Employee Salary : ",self.emp\_salary)

# method of a class

def displayEmployeeCount(self):

print("Employee Count : ",Employee.empCount)

emp1 = Employee('Shashank', 1000)

emp1.displayEmployeeInfo()

emp1.displayEmployeeCount()

emp2 = Employee('Rahul', 2000)

emp2.displayEmployeeInfo()

emp2.displayEmployeeCount()

emp1.displayEmployeeCount()

emp2.displayEmployeeCount()

Q7. What is the process for creating a class?

Ans:- class Employee:

# Constructor of class

# it is mainly used for assignment of instance variables

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

# instance variable or instance attributes

self.emp\_name = name

self.emp\_salary = salary

# method of a class

def displayEmployeeInfo(self):

print("Employee name : ",self.emp\_name, " , Employee Salary : ",self.emp\_salary)

emp1 = Employee('Shashank', 1000)

emp2 = Employee('Rahul', 2000)

emp1.displayEmployeeInfo()

emp2.displayEmployeeInfo()

print(emp1.emp\_name)

print(emp2.emp\_name)

Q8. How would you define the superclasses of a class?

Ans:- The class whose properties gets inherited by another class is known as **superclass or parent class**

**Two ways of initializing a superclass: explicitly using superclass name, or using super() method.**

1. Using the actual superclass name as: < SuperClassName >< . > \_\_init\_\_(self) . We initialized NumList this way in our first subclass.
2. Using a more general format as: < super() >< . > < \_\_init\_\_() >.

Q9. What is the relationship between classes and modules?

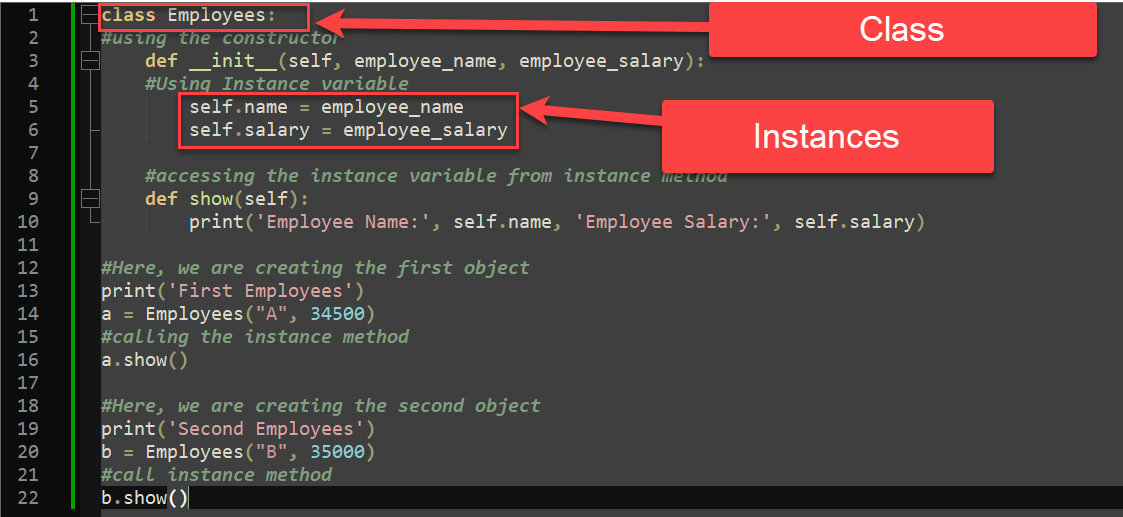
Ans:- Modules are collections of methods and constants. They cannot generate instances. Classes may generate instances (objects), and have per-instance state (instance variables).

Modules may be mixed in to classes and other modules. The mixed in module’s constants and methods blend into that class’s own, augmenting the class’s functionality. Classes, however, cannot be mixed in to anything.

A class may inherit from another class, but not from a module.

A module may not inherit from anything.

Q10. How do you make instances and classes?

Ans:- 

Q11. Where and how should be class attributes created?

Ans:- class Employee:

# Class attribute

empCount = 0

# Constructor of class

# it is mainly used for assignment of instance variables

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

# instance variable or instance attributes

self.emp\_name = name

self.emp\_salary = salary

Employee.empCount += 1

# method of a class

def displayEmployeeInfo(self):

print("Employee name : ",self.emp\_name, " , Employee Salary : ",self.emp\_salary)

# method of a class

def displayEmployeeCount(self):

print("Employee Count : ",Employee.empCount)

emp1 = Employee('Shashank', 1000)

emp1.displayEmployeeInfo()

emp1.displayEmployeeCount()

emp2 = Employee('Rahul', 2000)

emp2.displayEmployeeInfo()

emp2.displayEmployeeCount()

emp1.displayEmployeeCount()

emp2.displayEmployeeCount()

Q12. Where and how are instance attributes created?

Ans:- class Employee:

# Class attribute

empCount = 0

# Constructor of class

# it is mainly used for assignment of instance variables

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

# instance variable or instance attributes

self.emp\_name = name

self.emp\_salary = salary

Employee.empCount += 1

# method of a class

def displayEmployeeInfo(self):

print("Employee name : ",self.emp\_name, " , Employee Salary : ",self.emp\_salary)

# method of a class

def displayEmployeeCount(self):

print("Employee Count : ",Employee.empCount)

emp1 = Employee('Shashank', 1000)

emp1.displayEmployeeInfo()

emp1.displayEmployeeCount()

emp2 = Employee('Rahul', 2000)

emp2.displayEmployeeInfo()

emp2.displayEmployeeCount()

emp1.displayEmployeeCount()

emp2.displayEmployeeCount()

Q13. What does the term "self" in a Python class mean?

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

The reason you need to use self. is because Python does not use the @ syntax to refer to instance attributes. Python decided to do methods in a way that makes the instance to which the method belongs be passed automatically, but not received automatically: the first parameter of methods is the instance the method is called on.

In more clear way you can say that SELF has following Characteristic-

*Self is always pointing to Current Object.*

For example

class Employee:

# Constructor of class

# it is mainly used for assignment of instance variables

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

# instance variable or instance attributes

self.emp\_name = name

self.emp\_salary = salary

# method of a class

def displayEmployeeInfo(self):

print("Employee name : ",self.emp\_name, " , Employee Salary : ",self.emp\_salary)

emp1 = Employee('Shashank', 1000)

emp2 = Employee('Rahul', 2000)

emp1.displayEmployeeInfo()

emp2.displayEmployeeInfo()

print(emp1.emp\_name)

print(emp2.emp\_name)

Q14. How does a Python class handle operator overloading?

Ans:- In Python, overloading is achieved by **overriding the method which is specifically for that operator, in the user-defined class**. For example, \_\_add\_\_(self, x) is a method reserved for overloading + operator, and \_\_eq\_\_(self, x) is for overloading == .

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.

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

**Example:**

1. **class** example:
2. def \_\_init\_\_(self, X):
3. self.X = X
5. # adding two objects
6. def \_\_add\_\_(self, U):
7. **return** self.X + U.X
8. object\_1 = example( **int**( input( print ("Please enter the value: "))))
9. object\_2 = example( **int**( input( print ("Please enter the value: "))))
10. print (": ", object\_1 + object\_2)
11. object\_3 = example(str( input( print ("Please enter the value: "))))
12. object\_4 = example(str( input( print ("Please enter the value: "))))
13. print (": ", object\_3 + object\_4)

**Output:**

Please enter the value: 23

Please enter the value: 21

: 44

Please enter the value: Java

Please enter the value: Tpoint

: JavaTpoint

Q15. When do you consider allowing operator overloading of your classes?

Ans:- Consider that we have two objects which are a physical representation of a class (user-defined data type) and we have to add two objects with binary '+' operator it throws an error, because compiler don't know how to add two objects. So we define a method for an operator and that process is called operator overloading.

Q16. What is the most popular form of operator overloading?

Ans:- A very popular and convenient example is the **Addition (+) operator**. Just think how the '+' operator operates on two numbers and the same operator operates on two strings. It performs “Addition” on numbers whereas it performs “Concatenation” on strings.

Q17. What are the two most important concepts to grasp in order to comprehend Python OOP code?

## Ans:- Inheritance and Polymorphism

Object oriented programming (OOP) paradigm is built around the idea of having objects that belong to a particular type. In a sense, the type is what explains us the object.

Everything in Python is an object and every object has a type. These types are declared and defined using [classes](https://towardsdatascience.com/a-comprehensive-guide-for-classes-in-python-e6bb72a25a5e). Thus, classes can be considered as the heart of OOP.

In order to develop robust and well-designed software products with Python, it is essential to obtain a comprehensive understanding of OOP. In this article, we will elaborate on two key concepts of OOP which are inheritance and polymorphism.

Both inheritance and polymorphism are key ingredients for designing robust, flexible, and easy-to-maintain software. These concepts are best explained via examples. Let’s start with creating a simple class.

class Employee(): def \_\_init\_\_(self, emp\_id, salary):  
 self.emp\_id = emp\_id  
 self.salary = salary def give\_raise(self):  
 self.salary = self.salary \* 1.05

We have created a class called Employee. It has two data attributes which are employee id (emp\_id) and salary. We have also defined a method called give\_raise. It applies a 5-percent increase on the salary of an employee.

We can create an instance of the Employee class (i.e. an object with Employee type) and apply the give\_raise method as follows:

emp1 = Employee(1001, 56000)print(emp1.salary)  
56000emp1.give\_raise()print(emp1.salary)  
58800.0

OOP allows us to create a class based on another class. For instance, we can create the Manager class based on the Employee class.

class Manager(Employee):  
 pass

In this scenario, Manager is said to be a child class of the Employee class. The child class copies the attributes (both data and procedural) from the parent class. This concept is called **inheritance.**

It is important to note that inheritance does not mean copying a class. We can partially inherit from a parent (or base class). Python also allows for adding new attributes as well as modifying the existing ones. Thus, inheritance comes with a great deal of flexibility.

We can now create a manager object just like we create an employee object.

mgr1 = Manager(101, 75000)  
print(mgr1.salary)  
75000

A child class can have new attributes in addition to the ones inherited from the parent class. Furthermore, we have the option to modify or override the inherited attributes.

Let’s update the give\_raise method so that it applies a 10-percent increase for the managers.

class Manager(Employee): def give\_raise(self):  
 self.salary = self.salary \* 1.10mgr1 = Manager(101, 75000)  
print(mgr1.salary)  
75000mgr1.give\_raise()  
print(mgr1.salary)  
82500

We will create another child class of the Employee class. The Director class inherits the attributes from the Employee class and modifies the give\_raise method with a 20-percent increase.

class Director(Employee): def give\_raise(self):  
 self.salary = self.salary \* 1.20

We now have three different class and they all have a give\_raise method. Although the name of the method is the same, it behaves differently for different type of objects. This is an example of **polymorphism**.

Polymorphism allows for leveraging the same interface for different underlying operations. Regarding our example of manager and director objects, we can use them as they were an instance of the employee class.

Let’s see polymorphism in action. We will define a function that applies raise to a list of employees.

def bulk\_raise(list\_of\_emps):  
 for emp in list\_of\_emps:  
 emp.give\_raise()

The bulk\_raise function takes a list of employees and apply the give\_raise function to each object in the list. The next step is to create a list of employees of different types.

emp1 = Employee(101, 45000)  
emp2 = Manager(103, 60000)  
emp3 = Director(105, 70000)list\_of\_emps = [emp1, emp2, emp3]

Our list contains one employee, one manager, and one director objects. We can now call the bulk\_raise function.

bulk\_raise(list\_of\_emps)print(emp1.salary)  
47250.0print(emp2.salary)  
66000.0print(emp3.salary)  
84000.0

Although each object in the list has a different type, we do not have to explicitly state it in our function. Python knows which give\_raise method to apply.

As we see in the examples, polymorphism is accomplished using inheritance. Subclasses (or child classes) make use of the methods from the parent class to establish polymorphism.

## Conclusion

Both inheritance and polymorphism are fundamental concepts of object oriented programming. These concepts help us to create code that can be extended and easily maintainable.

Inheritance is a great way to eliminate unnecessary repetitive code. A child class can inherit from the parent class partially or entirely. Python is quite flexible with regards to inheritance. We can add new attributes and methods as well as modify the existing ones.

Polymorphism contributes to Python’s flexibility as well. An object with a particular type can be used as if it belonged to a different type. We have seen an example of it with the give\_raise method.

Q18. Describe three applications for exception processing.

Ans:- Exception processing is the process of **responding to unwanted or unexpected events when a computer program runs**. Exception processing deals with these events to avoid the program or system crashing, and without this process, exceptions would disrupt the normal operation of a program.

### The types of exceptions are

Exceptions can come in the following two exception classes:

1. **Checked exceptions.**Also called compile-time exceptions, the [compiler](https://www.techtarget.com/whatis/definition/compiler) checks these exceptions during the compilation process to confirm if the exception is being handled by the programmer. If not, then a compilation error displays on the system. Checked exceptions include SQLException and ClassNotFoundException.
2. **Unchecked exceptions.**Also called runtime exceptions, these exceptions occur during program execution. These exceptions are not checked at compile time, so the programmer is responsible for handling these exceptions. Unchecked exceptions do not give compilation errors. Examples of unchecked exceptions include NullPointerException and IllegalArgumentException.

The three applications for exception processing are

  specialized programming language constructs, [interrupt](https://www.techtarget.com/whatis/definition/interrupt) hardware mechanisms, and operating system [interprocess communication](https://www.techtarget.com/whatis/definition/interprocess-communication-IPC)

Q19. What happens if you don't do something extra to treat an exception?

Ans:- if you don't handle exceptions  
  
When an exception occurred, if you don't handle it, **the program terminates abruptly and the code past the line that caused the exception will not get executed**.

Q20. What are your options for recovering from an exception in your script?

Ans:- # How to handle exceptions in Python

# a = 10

# print("Hello !!")

# print( a/0 )

# print("Bye !!!")

# list\_a = [2,6,8,1,30]

# print( list\_a[7] )

# a = 5

# try:

# result = a/0

# print(result)

# except:

# print("Some Error Has Occured !!!")

# print("Byee !!!")

num = 5

list\_a = [1,2,3,4,7,90,20]

dict\_a = {'shashank' : 20, 'rahul' : 30}

try:

print("Divide number by 0 ")

#result = num/0

result = num/5

print(result)

print("Step 1 Done ")

print("Access 11'th element from List ")

# print(list\_a[11])

print(list\_a[5])

print("Step 2 Done ")

print("Access value of amit from dictionary ")

#print(dict\_a['amit'])

print(dict\_a['shashank'])

print("Step 3 Done ")

except ZeroDivisionError:

print("This Error Was Ocuured Because Division by 0 Happened !!")

except IndexError:

print("Error occured because out of bound index is getting accessed !!")

except KeyError:

print("Search Key Doesn't Exists !!")

except Exception as err:

print("Error Occured and Message : ", err)

# Use of Else Block

a = 5

try:

#result = a / 0

result = a / 2

except ZeroDivisionError:

print("Error Occured because of Division by 0 !!")

else:

print("Calculation completed !!")

print(result)

# Use Of Finally Keyword

a = 5

try:

result = a / 0

# result = a / 2

except ZeroDivisionError:

print("Error Occured because of Division by 0 !!")

else:

print("Calculation completed !!")

print(result)

finally:

print("Doesn't matter try-except but I will print myself !!")

**Handling Exceptions**

1. First, the try clause (the statement(s) between the try and except keywords) is executed.
2. If no exception occurs, the except clause is skipped and execution of the try statement is finished.
3. If an exception occurs during execution of the try clause, the rest of the clause is skipped.

Q21. Describe two methods for triggering exceptions in your script.

## Ans:- ****Exception Handling In Python****

In Python, exceptions can be handled by two new methods:

1. raise – You can use it to trigger an exception manually in your code
2. assert – You can use it to trigger an exception in your code conditionally

raise – You can use it to trigger an exception manually in your code.

### ****Raise Exceptions in Python****

To trigger exceptions, we need to code raise statements. Their general form is simple: the keyword, ‘raise’, followed by the name of the exception to be raised.  
Example:

class RaisingValueError(Exception):

def \_\_init\_\_(self, data):

self.data = data

def \_\_str\_\_(self):

return repr(self.data)Total\_movie = int(input(“Enter Total Movies Seen: “))

try:

Num\_of\_genres = int(input(“Enter Num of genres: “))

if(Num\_of\_genres < 1):

raise RaisingValueError(“Number of genres can’t be less than 1”)

except RaisingValueError as e:

print (“Try entering again:”, e.data)

Assert exception in python

You can use it to trigger an exception in your code conditionally

Now that you have enough theoretical knowledge of the concept let’s jump right into some code.

**Default Exception Handler**

Suppose we write the following code:

*#default exception handler*

test\_list = [1, 2, 3]

index = 5

print(test\_list[index])

**Copy code**

There is not much to this code; it simply tries to print a list element at a given index. Suppose the given index is greater than or equal to the length of the list, an exception will be triggered as Python detects out-of-bounds indexing operation for sequences and reports it by triggering the built-in IndexError exception.

You should know that exceptions have a rich type hierarchy. Because we have not explicitly mentioned any code to handle this exception, the default exception handler is invoked and a standard error message is displayed. A typical error message includes the exception that was raised, along with a list of code lines and functions that were active when the exception occurred, which we call ***stack trace***. The default handler also terminates the program immediately.

Q22. Identify two methods for specifying actions to be executed at termination time, regardless of  
whether or not an exception exists.

Ans:- try/finally – Allows to specify termination or cleanup actions, irrespective of exceptions occur or not.

Sometimes regardless of whether the exception has occurred or not, you want some action to occur. In such cases, you can use the **finally**clause with **try.**

For example, when you are working with files, you have to prevent more than one program from manipulating the same file at once. So, when a program finishes using a file, it should close the file to release it to other programs. Closing the file should be the cleanup action. You can achieve this by using **finally**.

*#Demonstrating try/finally clause*

test\_list = [1, 2, 3]

index = input('Enter the index: ')

try:

print(test\_list[index])

except:

print('The index should be between 0 &', len(test\_list) - 1, 'but was given', index)

finally:

print("\nFinally Block: This is always executed")

**Copy code**

Q23. What is the purpose of the try statement?

Ans:- The try block lets you test a block of code for errors. The try statement **allows you to define a block of code to be tested for errors while it is being executed**.

Q24. What are the two most popular try statement variations?

Ans:- There are two other optional segments to a try block: **else and finally** . Both of these optional blocks will come after the try and the except . Also, there's nothing stopping you from using both else and finally in a single statement — but keep them in that order if you do.

# The Different Try/Except Variations

So far we’ve used a try/except and even a try/except/except, but this is only two-thirds of the story.

There are two other optional segments to a try block: else and finally. Both of these optional blocks will come **after** the try and the except. Also, there’s nothing stopping you from using both else and finally in a single statement — but keep them in that order if you do.

Let’s go through each individually and see how they extend the behavior of a simple try/except.

## Try/Except/Else

When attaching an else statement to the end of a try/except, this code will be executed **after** the try has been completed, but only **if no exceptions occur**.

We can take the previous example of prompting a user for an integer input and use an else block to thank them for valid input and breaking out of the while loop.

while True:  
 try:  
 num = int(input("Enter an int: "))  
 except Exception as e:  
 print(e)  
 else:  
 print("Thank you for the integer!")  
 break# Enter an int: a  
# invalid literal for int() with base 10: 'a'  
# Enter an int: 3  
# Thank you for the integer

## Try/Except/Finally

When attaching a finally statement to the end of a try/except, this code will be executed **after** the try has been completed, **regardless of exceptions**.

Again, we’ll use our previous example and add a simple counter to illustrate this behavior.

count = 0  
while True:  
 try:  
 num = int(input("Enter an int: "))  
 break  
 except Exception as e:  
 print(e)  
 finally:  
 count += 1  
 print("Attempt #:",count)# Enter an int: a  
# invalid literal for int() with base 10: 'a'  
# Attempt #: 1  
# Enter an int: 3  
# Attempt #: 2

This might look a bit odd because the break is still inside the try. It’s reasonable to think that the finally would be cut short upon proper input, however, that’s not the case. The finally section will still execute, regardless of how the try is exited.

It’s a very small next step to take what we’ve demonstrated here and form a try/except/else/finally statement.

Q25. What is the purpose of the raise statement?

Ans:- Python raise Keyword is used **to raise exceptions or errors**. The raise keyword raises an error and stops the control flow of the program. It is used to bring up the current exception in an exception handler so that it can be handled further up the call stack.

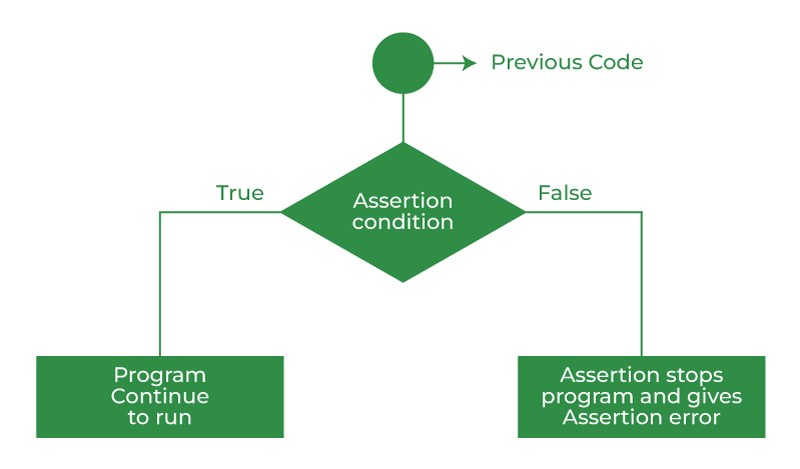
Q26. What does the assert statement do, and what other statement is it like?

Ans:- The assert keyword is used when debugging code. The assert keyword lets you test if a condition in your code returns True, if not, the program will raise an AssertionError. You can write a message to be written if the code returns False, check the example below. assert statement takes an expression and optional message. assert statement is used **to check types, values of argument and the output of the function**. assert statement is used as debugging tool as it halts the program at the point where an error occurs.

**assert in Python**

In simpler terms, we can say that assertion is the boolean expression that checks if the statement is True or False. If the statement is true then it does nothing and continues the execution, but if the statement is False then it stops the execution of the program and throws an error.

**Flowchart of Python Assert Statement**



*assert in Python*

**assert Keyword in Python**

In python, **assert** keyword helps in achieving this task. This statement takes as input a boolean condition, which when returns true doesn’t do anything and continues the normal flow of execution, but if it is computed to be false, then it raises an AssertionError along with the optional message provided.

***Syntax :****assert condition, error\_message(optional)*

Q27. What is the purpose of the with/as argument, and what other statement is it like?

Ans:- **The with statement works with the open() function to open a file**. The **with statement** in Python is used for resource management and exception handling. You’d most likely find it when working with file streams. For example, the statement ensures that the file stream process doesn’t block other processes if an exception is raised, but terminates properly.

In Python, the with statement **replaces a try-catch block with a concise shorthand**. More importantly, it ensures closing resources right after processing them. A common example of using the with statement is reading or writing to a file.

A function or class that supports the with statement is known as a context manager. A context manager allows you to open and close resources right when you want to. For example, the open() function is a context manager. When you call the open() function using the with statement, the file closes automatically after you’ve processed the file.

The with statement is a replacement for commonly used try/finally error-handling statements. A common example of using the with statement is opening a file. To open and write to a file in Python, you can use the with statement as follows:

with open("example.txt", "w") as file:

file.write("Hello World!")

The with statement automatically closes the file after you’ve completed writing it.

Under the hood, the with statement replaces this kind of try-catch block:

f = open("example.txt", "w")

try:

f.write("hello world")

finally:

f.close()

Q28. What are \*args, \*\*kwargs?

Ans:- **Special Symbols Used for passing arguments:-**

* \*args (Non-Keyword Arguments)
* \*\*kwargs (Keyword Arguments)

***Note:****“We use the “wildcard” or “\*” notation like this – \*args OR \*\*kwargs – as our function’s argument when we have doubts about the number of  arguments we should pass in a function.”*

**Python \*args**

The special syntax *\*args* in function definitions in python is used to pass a variable number of arguments to a function. It is used to pass a non-key worded, variable-length argument list.

* The syntax is to use the symbol \* to take in a variable number of arguments; by convention, it is often used with the word args.
* What *\*args* allows you to do is take in more arguments than the number of formal arguments that you previously defined. With *\*args*, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).
* For example, we want to make a multiply function that takes any number of arguments and is able to multiply them all together. It can be done using \*args.
* Using the \*, the variable that we associate with the \* becomes an iterable meaning you can do things like iterate over it, run some higher-order functions such as map and filter, etc.

**Example 1:**

Python program to illustrate \*args for a variable number of arguments

|  |
| --- |
| def myFun(\*argv):      for arg in argv:          print(arg)      myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks') |

**Output:**

Hello

Welcome

to

GeeksforGeeks

**Example 2:**

Python program to illustrate \*args with a first extra argument

* Python3

|  |
| --- |
| def myFun(arg1, \*argv):      print("First argument :", arg1)      for arg in argv:          print("Next argument through \*argv :", arg)      myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks') |

Output:- First argument : Hello

Next argument through \*argv : Welcome

Next argument through \*argv : to

Next argument through \*argv : GeeksforGeeks

**What is Python \*\*kwargs**

The special syntax *\*\*kwargs* in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name *kwargs* with the double star. The reason is that the double star allows us to pass through keyword arguments (and any number of them).

* A keyword argument is where you provide a name to the variable as you pass it into the function.
* One can think of the *kwargs* as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the *kwargs* there doesn’t seem to be any order in which they were printed out.

**Example 1:**

Python program to illustrate \*kwargs for a variable number of keyword arguments. Here \*\*kwargs accept keyworded variable-length argument passed by the function call. for first=’Geeks’ first is key and ‘Geeks’ is a value. in simple words, what we assign is value, and to whom we assign is key.

|  |
| --- |
| def myFun(\*\*kwargs):      for key, value in kwargs.items():          print("%s == %s" % (key, value))      # Driver code  myFun(first='Geeks', mid='for', last='Geeks') |

**Output:**

first == Geeks

mid == for

last == Geeks

**Example 2:**

Python program to illustrate  \*\*kwargs for a variable number of keyword arguments with one extra argument. All the same, but one change is we passing non-keyword argument which acceptable by positional argument(arg1 in myFun). and keyword arguments we passing are acceptable by \*\*kwargs. simple right?

* Python3

|  |
| --- |
| def myFun(arg1, \*\*kwargs):      for key, value in kwargs.items():          print("%s == %s" % (key, value))      # Driver code  myFun("Hi", first='Geeks', mid='for', last='Geeks') |

**Output:**

first == Geeks

mid == for

last == Geeks

**Using both \*args and \*\*kwargs to call a function**

**Example 1:**

Here, we are passing \*args and \*\*kwargs as an argument in the myFun function. By passing \*args to myFun simply means that we pass the positional and variable-length arguments which are contained by args. so, “Geeks” pass to the arg1 , “for” pass to the arg2, and “Geeks” pass to the arg3. When we pass \*\*kwargs as an argument to the myFun it means that it accepts keyword arguments. Here, “arg1” is key and the value is “Geeks” which is passed to arg1, and just like that “for” and “Geeks” pass to arg2 and arg3 respectively. After passing all the data we are printing all the data in lines.

* python3

|  |
| --- |
| def myFun(arg1, arg2, arg3):      print("arg1:", arg1)      print("arg2:", arg2)      print("arg3:", arg3) |
| # Now we can use \*args or \*\*kwargs to  # pass arguments to this function :  args = ("Geeks", "for", "Geeks")  myFun(\*args)    kwargs = {"arg1": "Geeks", "arg2": "for", "arg3": "Geeks"}  myFun(\*\*kwargs) |

**Output:**

arg1: Geeks

arg2: for

arg3: Geeks

arg1: Geeks

arg2: for

arg3: Geeks

**Example 2:**

Here, we are passing \*args and \*\*kwargs as an argument in the myFun function. where **‘geeks’, ‘for’, ‘geeks’** is passed as \*args, and **first=”Geeks”, mid=”for”, last=”Geeks”** is passed as \*\*kwargs and printing **in the same line.**

|  |
| --- |
| def myFun(\*args, \*\*kwargs):      print("args: ", args)      print("kwargs: ", kwargs)      # Now we can use both \*args ,\*\*kwargs  # to pass arguments to this function :  myFun('geeks', 'for', 'geeks', first="Geeks", mid="for", last="Geeks") |

**Output:**

args: ('geeks', 'for', 'geeks')

kwargs: {'first': 'Geeks', 'mid': 'for', 'last': 'Geeks'}

**Using \*args and \*\*kwargs to set values of object**

* \*args receives arguments as a **tuple**.
* \*\*kwargs receives arguments as a **dictionary.**
* class car(): #defining car class
* def \_\_init\_\_(self,\*args): #args receives unlimited no. of arguments as an array
* self.speed = args[0] #access args index like array does
* self.color=args[1]
* #creating objects of car class
* audi=car(200,'red')
* bmw=car(250,'black')
* mb=car(190,'white')
* print(audi.color)
* print(bmw.speed)

**Output**

red

250

With \*\*kwargs

class car(): #defining car class

    def \_\_init\_\_(self,\*\*kwargs): #args receives unlimited no. of arguments as an array

        self.speed = kwargs['s'] #access args index like array does

        self.color = kwargs['c']

#creating objects of car class

audi=car(s=200,c='red')

bmw=car(s=250,c='black')

mb=car(s=190,c='white')

print(audi.color)

print(bmw.speed)

**Output**

red

250

Q29. How can I pass optional or keyword parameters from one function to another?

Ans:- To pass optional or keyword parameters from one function to another, collect the arguments using the \* and \*\* specifiers in the function’s parameter list But, at first, do know what are \*args and \*\*args in Python. Let us understand them −

## Variable-length/ Arbitrary arguments in Python (\*args)

### Example

When you don’t know in advance about the number of arguments to be passed, the arguments are variable-length. Include an asterisk i.e. \* before the parameter name while defining the function. Let us see an example:

def demo(\*car):

print("Car 1 = ",car[0])

print("Car 2 = ",car[1])

print("Car 3 = ", car[2])

print("Car 4 = ", car[3])

# call

demo("Tesla", "AudTo pass optional or keyword parameters from one function to another, collect the arguments using the \* and \*\* specifiers in the function’s parameter list But, at first, do know what are \*args and \*\*args in Python. Let us understand them −

Variable-length/ Arbitrary arguments in Python (\*args)

Example

When you don’t know in advance about the number of arguments to be passed, the arguments are variable-length. Include an asterisk i.e. \* before the parameter name while defining the function. Let us see an example:

i", "BMW", "Toyota")

### Output

('Car 1 = ', 'Tesla')

('Car 2 = ', 'Audi')

('Car 3 = ', 'BMW')

('Car 4 = ', 'Toyota')

## Arbitrary Keyword Arguments in Python (\*\*kwargs)

When you don’t know in advance about the number of keyword arguments to be passed, the arguments are arbitrary keyword arguments.

### Example

Let us see an example −

def demo(\*\*c):

print("Car Name: "+c["name"])

print("Car Model: "+c["model"])

# call

demo(name = "Tesla", model = "2022")

### Output

Car Name: Tesla

Car Model: 2022

Pass optional or keyword parameters from one function to another

To pass, collect the arguments using the \* and \*\* in the function’s parameter list. Through this, you will get the positional arguments as a tuple and the keyword arguments as a dictionary. Pass these arguments when calling another function by using \* and \*\* −

def f(a, \*args, \*\*kwargs):

...

kwargs['width'] = '14.3c'

...

g(a, \*args, \*\*kwargs)

Q30. What are Lambda Functions?

Ans:- A lambda function is an anonymous function (i.e., defined without a name) that can take any number of arguments but, unlike normal functions, evaluates and returns only one expression.

A lambda function in Python has the following syntax:

lambda parameters: expression

The anatomy of a lambda function includes three elements:

* The **keyword lambda** — an analog of def in normal functions
* The **parameters** — support passing positional and keyword  
  arguments, just like normal functions
* The **body** — the expression for given parameters being evaluated  
  with the lambda function

Note that, unlike a normal function, we don’t surround the parameters of a lambda function with parentheses. If a lambda function takes two or more parameters, we list them with a comma.

We use a lambda function to evaluate only one short expression (ideally, a single-line) and only once, meaning that we aren’t going to apply this function later. Usually, we pass a lambda function as an argument to a higher-order function (the one that takes in other functions as arguments), such as Python built-in functions like filter(), map(), or reduce().

## How a Lambda Function in Python Works

Let’s look at a simple example of a lambda function:

lambda x: x + 1

<function \_\_main\_\_.<lambda>(x)>

The lambda function above takes a single argument, increments it by 1, and returns the result. It’s a simpler version of the following normal function with the def and return keywords:

def increment\_by\_one(x):

return x + 1

For now, however, our lambda function lambda x: x + 1 only creates a function object and doesn’t return anything. We expected this: we didn’t provide any value (an argument) to its parameter x. Let’s assign a variable first, pass it to the lambda function, and see what we get this time:

a = 2

print(lambda x: a + 1)

<function <lambda> at 0x00000250CB0A5820>

Instead of returning 3, as we might expect, our lambda function returned the function object itself and its memory location. Indeed, this isn’t the right way to call a lambda function. To pass an argument to a lambda function, execute it, and return the result, we should use the following syntax:

(lambda x: x + 1)(2)

3

Note that while the parameter of our lambda function isn’t surrounded with parentheses, when we call it, we add parentheses around the entire construction of the lambda function and around the argument we passed to it.

Another thing to notice in the code above is that with a lambda function, we can execute the function immediately after its creation and receive the result. This is the so-called **immediately invoked function execution** (or **IIFE**).

We can create a lambda function with multiple parameters. In this case, we separate the parameters in the function definition with a comma. When we execute such a lambda function, we list the corresponding arguments in the same order and separate them with a comma, too:

(lambda x, y, z: x + y + z)(3, 8, 1)

12

It’s also possible to use a lambda function to perform conditional operations. Below is a lambda analog for a simple if-else function:

print((lambda x: x if(x > 10) else 10)(5))

print((lambda x: x if(x > 10) else 10)(12))

10

12

If multiple conditions are present (if-elif-…-else), we have to nest them:

(lambda x: x \* 10 if x > 10 else (x \* 5 if x < 5 else x))(11)

110

The issue with this approach is that already with one nested condition, the code becomes difficult to read, as we can see above. In such situations, a normal function with an if-elif-…-else set of conditions would be a better choice than a lambda function. Indeed, we can write the lambda function from the example above in the following way:

def check\_conditions(x):

if x > 10:

return x \* 10

elif x < 5:

return x \* 5

else:

return x

check\_conditions(11)

110

Even though the function above spans more lines than the corresponding lambda function, it’s much easier to read.

We can assign a lambda function to a variable and then call that variable as a normal function:

increment = lambda x: x + 1

increment(2)

3

However, it’s a bad practice, according the PEP 8 style guide for Python code:

The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit def statement (i.e., that it can be embedded inside a larger expression).

So, if we really need to store a function for further usage, instead of assigning a lambda function to a variable, we’d better define an equivalent normal function.

Q31. Explain Inheritance in Python with an example?

# Ans:- 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

1. **class** derived-**class**(base **class**):
2. <**class**-suite>

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

### Syntax

1. **class** derive-**class**(<base **class** 1>, <base **class** 2>, ..... <base **class** n>):
2. <**class** - suite>

### Example 1

1. **class** Animal:
2. **def** speak(self):
3. **print**("Animal Speaking")
4. #child class Dog inherits the base class Animal
5. **class** Dog(Animal):
6. **def** bark(self):
7. **print**("dog barking")
8. d = Dog()
9. d.bark()
10. d.speak()

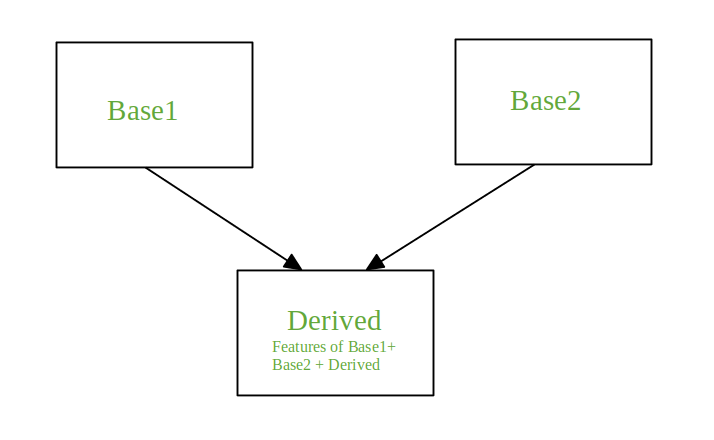
**Output:**

dog barking

Animal Speaking

Q32. Suppose class C inherits from classes A and B as class C(A,B).Classes A and B both have their own versions of method func(). If we call func() from an object of class C, which version gets invoked?

Ans:- **Multiple Inheritance**   
When a class is derived from more than one base class it is called multiple Inheritance. The derived class inherits all the features of the base case.



**Syntax:**

Class Base1:

Body of the class

Class Base2:

Body of the class

Class Derived(Base1, Base2):

Body of the class

Class A:

Body of the class

Class B:

Body of the class

Class C(A,B):

Body of the class

For example:-

class Rectangle:

def \_\_init\_\_(self, length, width, \*\*kwargs):

self.length = length

self.width = width

super().\_\_init\_\_(\*\*kwargs)

def area(self):

return self.length \* self.width

def perimeter(self):

return 2 \* self.length + 2 \* self.width

class Square(Rectangle):

def \_\_init\_\_(self, length, \*\*kwargs):

super().\_\_init\_\_(length=length, width=length, \*\*kwargs)

class Triangle:

def \_\_init\_\_(self, base, height, \*\*kwargs):

self.base = base

self.height = height

super().\_\_init\_\_(\*\*kwargs)

def tri\_area(self):

return 0.5 \* self.base \* self.height

class RightPyramid(Square, Triangle):

def \_\_init\_\_(self, base, slant\_height, \*\*kwargs):

self.base = base

self.slant\_height = slant\_height

kwargs["height"] = slant\_height

kwargs["length"] = base

super().\_\_init\_\_(base=base, \*\*kwargs)

def area(self):

base\_area = super().area()

perimeter = super().perimeter()

return 0.5 \* perimeter \* self.slant\_height + base\_area

def area\_2(self):

base\_area = super().area()

triangle\_area = super().tri\_area()

return triangle\_area \* 4 + base\_area

Q33. Which methods/functions do we use to determine the type of instance and inheritance?

Ans:- **Python has two built-in functions that work with inheritance:**

* Use isinstance() to check an instance's type: isinstance(obj, int) will be True only if obj.\_\_class\_\_ is int or some class derived from int .
* Use issubclass() to check class inheritance: issubclass(bool, int) is True since bool is a subclass of int .

# isinstance() and issubclass()

The **isinstance()** method checks whether an object is an instance of a class whereas **issubclass()** method asks whether one class is a subclass of another class (or other classes).

### isinstance(object, classinfo)

Return true if the object argument is an instance of the **classinfo** argument, or of a (direct, indirect or virtual) subclass thereof.

### issubclass(class, classinfo)

Return true if class is a **subclass** (direct, indirect or virtual) of classinfo. A class is considered a subclass of itself.

#### example

class MyClass(object):

pass

class MySubClass(MyClass):

pass

print(isinstance(MySubClass, object))

print(issubclass(MySubClass, MyClass))

print(isinstance(MySubClass, MyClass))

#### output

True

True

False

Q34.Explain the use of the 'nonlocal' keyword in Python.

Ans:- The nonlocal keyword is used to work with variables inside nested functions, where the variable should not belong to the inner function. Use the keyword nonlocal to declare that the variable is not local.

Q35. What is the global keyword?

# Ans:- Python Global Keyword (With Examples)

Many a time we are required to declare a variable that is to be used for multiple times in our program. But its value can keep changing according to the situation and the function it is being used for. Hence, [Python](https://www.toppr.com/guides/computer-science/introduction-to-python/getting-started-with-python/python/) provides us a method to declare a variable globally, which can be accessed within class and functions, and the value of it can be changed only for that class or function respectively. Python global keyword is used to declare that variable in the program. This article will learn more about the Python global keyword, its syntax, when to use it, and examples for proper understanding.

## What is the Global Keyword?

* **Python global keyword** allows the user to define a variable globally whose value can be modified outside the current scope.

The term global denotes that the scope of a variable declared as global will persist for the duration of the entire program’s execution. We utilize the global keyword inside a function only when we wish to conduct value assignments or update a variable.

## Rules of Global Keyword

Few basic rules for using the Python global keyword:

* Without the use of the global keyword, a variable created inside any function is a Local variable by default.
* Variable defined outside the function is Global by default. No need to use the global keyword.
* Global keyword is used when we want to read or write any global variable value inside the function.
* The global keyword used for a variable declared outside the function does not have any effect on it.
* In the same line, a variable cannot be declared global and assigned a value. E.g. global x = 5 is not allowed.

## Use of Global Keyword

In a Python program, distinct variables have different scopes. The variable may or may not be available inside a function, depending on where it is declared. We may need to change a variable inside a function from outside its current scope on occasion. In this case, we utilize the global keyword in conjunction with the variable name.

In Python, the global keyword allows you to change a variable value outside of its current scope. It is used to make changes to a global variable from a local location. The global keyword is only required for altering the variable value and not for publishing or accessing it.

Let us understand various examples where can we use the global keyword.

#### ****Accessing Global variable from inside the function****

In this example, the variable is declared outside the function i.e., globally. We will simply fetch the variable into the local function and print it.

Example 1 –

COPY CODE

# Python program to illustrate the usage of global variable in a local function

num = 25 # Global variable

def display():

print('Global Variable value inside function:', num)

display()

Output –

Global Variable value inside function: 25