**UNIT – 3**

**Procedural Programming vs Object-Oriented Programming**

| **Procedural Oriented Programming** | **Object-Oriented Programming** |
| --- | --- |
| In procedural programming, the program is divided into small parts called ***functions***. | In object-oriented programming, the program is divided into small parts called ***objects***. |
| Procedural programming follows a ***top-down approach***. | Object-oriented programming follows a ***bottom-up approach***. |
| There is **no access specifier** in procedural programming. | Object-oriented programming has access specifiers like **private, public, protected**, etc. |
| Adding new data and functions is **not easy**. | Adding new data and function is **easy**. |
| Procedural programming does not have any proper way of hiding data so it is ***less secure***. | Object-oriented programming provides data hiding so it is ***more secure***. |
| In procedural programming, **overloading** is **not possible**. | **Overloading is possible** in object-oriented programming. |
| In procedural programming, there is **no** concept of **data hiding and inheritance**. | In object-oriented programming, the concept of **data hiding and inheritance** is used. |
| In procedural programming, the **function** is **more important** than the data. | In object-oriented programming, **data** is **more important** than function. |
| Procedural programming is based on the ***unreal world***. | Object-oriented programming is based on the ***real world***. |
| Procedural programming is used for **designing medium-sized programs**. | Object-oriented programming is used for **designing large and complex programs**. |
| Procedural programming uses the concept of **procedure abstraction**. | Object-oriented programming uses the concept of **data abstraction**. |
| **Code reusability absent** in procedural programming, | **Code reusability present** in object-oriented programming. |
| **Examples:** C, FORTRAN, Pascal, Basic, etc. | **Examples:**C++, Java, Python, C#, etc. |

**PYTHON OOPS**

In Python, object-oriented Programming (OOPs) is a programming paradigm that **uses objects and classes in programming**. It aims to **implement real-world entities** like inheritance, polymorphisms, encapsulation, etc. in the programming. **The** **main concept of OOPs is** **to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data**.

## OOPs Concepts in Python

* Class
* Objects
* Polymorphism
* Encapsulation
* Inheritance
* Data Abstraction

## **Python Class**

A class is a **collection of objects**. A class contains the **blueprints or the prototype** from which the objects are being created. It is a **logical entity** that contains some **attributes and methods**.

To understand the need for creating a class let’s consider an example, let’s say you wanted to track the number of dogs that may have different attributes like breed, and age. If a list is used, the first element could be the dog’s breed while the second element could represent its age. Let’s suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it’s the exact need for classes.

**Some points on Python class:**

* Classes are created by **keyword class**.
* **Attributes** are the **variables that belong to a class**.
* **Attributes** are **always public** and can be **accessed** using the **dot (.) operator**. Eg.: **Myclass.Myattribute**

**Class Definition Syntax:**

class ClassName:  
 # Statement-1  
 .  
 .  
 .  
 # Statement-N

### Creating an Empty Class in Python

In this example, we have created a class named Dog using the class keyword.

|  |
| --- |
| # Python3 program to  # demonstrate defining  # a class    **class** Dog:  **pass** |

## **Python Objects**

The object is an **entity** that has a **state and behavior** **associated with it**. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number 12 is an object, the string “Hello, world” is an object, a list is an object that can hold other objects, and so on. You’ve been using objects all along and may not even realize it.

**An object consists of:**

* **State:** It is represented by the **attributes of an object**. It also reflects the properties of an object.
* **Behavior:** It is represented by the **methods of an object**. It also reflects the response of an object to other objects.
* **Identity:** It gives a **unique name to an object** and enables one object to interact with other objects.

To understand the state, behavior, and identity let us take the example of the class dog (explained above).

* The identity can be considered as the name of the dog.
* State or Attributes can be considered as the breed, age, or color of the dog.
* The behavior can be considered as to whether the dog is eating or sleeping.

### Creating an Object

This will create an object named obj of the class Dog defined above. Before diving deep into objects and classes let us understand some basic keywords that will we used while working with objects and classes.

|  |
| --- |
| obj **=** Dog() |

### ****The Python self****

1. **Class methods must have an extra first parameter in the method definition**. We do not give a value for this parameter when we call the method, Python provides it.
2. If we have a **method that takes no arguments**, then we **still have to pass self** as one argument.
3. This is similar to **this pointer in C++** and **this reference in Java**.

When we call a method of this object as **myobject.method(arg1, arg2),** this is **automatically converted by Python** into **MyClass.method(myobject, arg1, arg2)** – this is all the special self is about.

### ****The Python \_\_init\_\_ Method****

The [\_\_init\_\_ method](https://www.geeksforgeeks.org/__init__-in-python/) is similar to **constructors in C++ and Java**. It is **run as soon as** **an object of a class is instantiated**. The method is **useful to do any initialization you want to do with your object**. Now let us define a class and create some objects using the self and \_\_init\_\_ method.

### Creating a class and object with class and instance attributes

|  |
| --- |
| **class** Dog:        # class attribute      attr1 **=** "mammal"        # Instance attribute  **def** \_\_init\_\_(self, name):          self.name **=** name    # Driver code  # Object instantiation  Rodger **=** Dog("Rodger")  Tommy **=** Dog("Tommy")    # Accessing class attributes  print("Rodger is a {}".format(Rodger.\_\_class\_\_.attr1))  **print**("Tommy is also a {}".format(Tommy.\_\_class\_\_.attr1))    # Accessing instance attributes  print("My name is {}".format(Rodger.name))  **print**("My name is {}".format(Tommy.name)) |

**Output**

Rodger is a mammal

Tommy is also a mammal

My name is Rodger

My name is Tommy

### Creating Classes and objects with methods

Here, The Dog class is defined with two attributes:

* attr1 is a class attribute set to the value “mammal”. **Class attributes are shared by all instances of the class**.
* \_\_init\_\_ is a special method (constructor) that **initializes an instance of the Dog class**. It takes two parameters: self (referring to the instance being created) and name (representing the name of the dog). The name parameter is used to assign a name attribute to each instance of Dog.
* The speak method is defined within the Dog class. This method prints a string that includes the name of the dog instance.

The driver code starts by creating two instances of the Dog class: Rodger and Tommy. The \_\_init\_\_ method is called for each instance to initialize their name attributes with the provided names. The speak method is called in both instances (Rodger.speak() and Tommy.speak()), causing each dog to print a statement with its name.

|  |
| --- |
| **class** Dog:        # class attribute      attr1 **=** "mammal"        # Instance attribute  **def** \_\_init\_\_(self, name):          self.name **=** name    **def** speak(self):  **print**("My name is {}".format(self.name))    # Driver code  # Object instantiation  Rodger **=** Dog("Rodger")  Tommy **=** Dog("Tommy")    # Accessing class methods  Rodger.speak()  Tommy.speak() |

**Output**

My name is Rodger

My name is Tommy

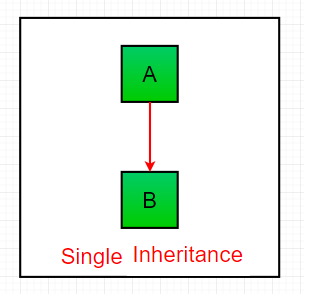
## **(a) Python Inheritance -**

**Inheritance is the capability of one class to derive or inherit the properties from another class.** The class that derives properties is called the **derived class or 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:

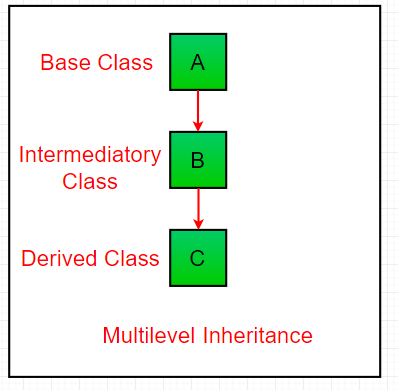
* It **represents real-world relationships** well.
* It provides the **reusability of a code**. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
* It is **transitive in nature**, which means that if class B inherits from another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

#### **Types of Inheritance**

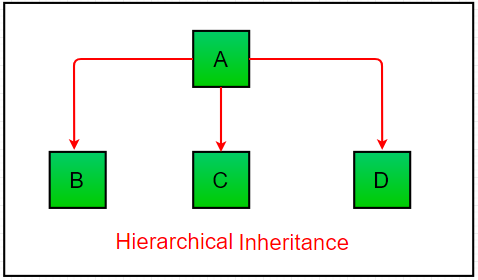
* **Single Inheritance**: Single-level inheritance enables a derived class to inherit characteristics from a **single-parent class**.



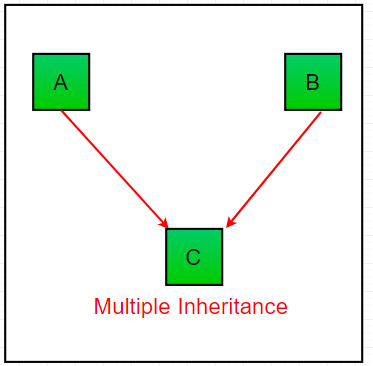
* **Multilevel Inheritance:**Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.



* **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 code to demonstrate how parent constructors are called.    # parent class  **class** Person(object):        # \_\_init\_\_ is known as the constructor  **def** \_\_init\_\_(self, name, idnumber):          self.name **=** name          self.idnumber **=** idnumber    **def** display(self):          print(self.name)          print(self.idnumber)    **def** details(self):  **print**("My name is {}".format(self.name))          print("IdNumber: {}".format(self.idnumber))    # 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()  a.details() |

**Output**

Rahul

886012

My name is Rahul

IdNumber: 886012

Post: Intern

## **Python Polymorphism**

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

### Polymorphism in Python

This code demonstrates the concept of **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.")    **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() |

**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.

## **Data Abstraction**

It **hides unnecessary code details from the user**. Also,  when we do not want to give out sensitive parts of our code implementation and this is where data abstraction came.

Data Abstraction in Python can be achieved by **creating abstract classes**.

## **Python Encapsulation**

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of **wrapping data and the methods that work on data within one unit**. This **puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data**. **To prevent accidental change, an object’s variable can only be changed by an object’s method.** Those types of variables are known as **private variables**.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

**Encapsulation in Python**

In the above example, we have created the c variable as the private attribute. We cannot even access this attribute directly and can’t even change its value.

|  |
| --- |
| # Python program to  # demonstrate private members  # **"\_\_" double underscore represents private attribute.**  # **Private attributes start with "\_\_".**    # Creating a Base class  **class** Base:  **def** \_\_init\_\_(self):          self.a **=** "GeeksforGeeks"          self.\_\_c **=** "GeeksforGeeks"    # Creating a derived class  **class** Derived(Base):  **def** \_\_init\_\_(self):           # Calling constructor of          # Base class          Base.\_\_init\_\_(self)  **print**("Calling private member of base class: ")          print(self.\_\_c)   # Driver code  obj1 **=** Base()  print(obj1.a)   # Uncommenting print(obj1.c) will  # raise an AttributeError   # Uncommenting obj2 = Derived() will  # also raise an AtrributeError as  # private member of base class  # is called inside derived class |

**Output**

GeeksforGeeks

**INSTANCE VARIABLE / ATTRIBUTE**

**Instance attributes** are those attributes that are **not shared by objects**. Every object has its own copy of the instance attribute i.e. **for every object, instance attribute is different. Instance variables are *unique to each instance of a class***. They are defined within methods and are **prefixed** with the **self keyword**. These variables **store data** that is **specific to an instance**, making them essential for object-oriented programming (OOP) principles like encapsulation. **There are two ways to access the instance variable of class:**

* Within the class by using **self** and object reference.
* Using **getattr()** method

**Example 1:** **Using Self and object reference**

|  |
| --- |
| #creating class  **class** student:        # constructor  **def** \_\_init\_\_(self, name, rollno):            # instance variable          self.name **=** name          self.rollno **=** rollno    **def** display(self):            # using self to access          # variable inside class  **print**("hello my name **is**:", self.name)          print("my roll number **is**:", self.rollno)    # Driver Code  # object created  s **=** student('HARRY', 1001)    # function call through object  s.display()    # accessing variable from  # outside the class  **print**(s.name) |

**Output:**

hello my name is: HARRY

my roll number is: 1001

HARRY

**Example 2:** **Using getattr()**

|  |
| --- |
| # Python code for accessing attributes of class  **class** emp:      name**=**'Harsh'      salary**=**'25000'  **def** show(self):          print(self.name)          print(self.salary)    # Driver Code  e1 **=** emp()  # Use getattr instead of e1.name  print(getattr(e1,'name'))    # returns true if object has attribute  print(hasattr(e1,'name'))    # sets an attribute  setattr(e1,'height',152)    # returns the value of attribute name height  print(getattr(e1,'height'))    # delete the attribute  delattr(emp,'salary') |

**Output:**

Harsh

True

152

## **CLASS VARIABLES**

**Class variables** are ***shared among all instances of a class***. They are **defined within the class but outside of any methods**, typically **near the top of the class definition**. Class variables **store data that is common to all instances**, making them a powerful tool for managing shared state and settings.

class Dog:  
 species = "Canis familiaris" # Class variable  
 def \_\_init\_\_(self, name):  
 self.name = name # Instance variable  
dog1 = Dog("Buddy")  
dog2 = Dog("Milo")  
print(dog1.species) # Output: "Canis familiaris"  
print(dog2.species) # Output: "Canis familiaris"

**CLASS VARIABLE VS INSTANCE VARIABLE**

| **Instance Variable** | **Class Variable** |
| --- | --- |
| It is a variable whose value is **instance-specific and not shared among instances.** | It is a variable that defines a **specific attribute or property for a class.** |
| These variables **cannot be shared between classes.** Instead, they only belong to one specific class. | These variables can be **shared between class and its subclasses.** |
| It usually **reserves memory for data** that the class needs. | It usually **maintains a single shared value** for all instances of class even if no instance object of the class exists. |
| It is generally **created when an instance of the class is created.** | It is generally **created when the program begins to execute.** |
| It normally retains values **as long as the object exists.** | It normally retains values **until the program terminates.** |
| It has **many copies so every object has its own personal copy** of the instance variable. | It has **only one copy of the class variable** so it is shared among different objects of the class. |
| It can be **accessed directly by calling variable names inside the class**. | It can be **accessed by calling with the class name**. |
| These variables are **declared without using the static keyword**. | These variables **are declared using the static keyword**. |
| **Changes** that are made to these variables through one object **will not reflect in another object.** | **Changes** that are made to these variables through one object **will reflect in another object**. |

**INSTANCE METHOD**

An **Instance Method** is a **function that works with a class instance**. An instance method can **access and even modify the value of attributes of an instance**. The instance methods are **bound to the class instance** and perform a set of actions on the data/value given by the object (instance) variables**. If we use the instance variable inside the methods, these methods are called instance methods**. It can **modify the object state**. It has **one default parameter** :- **self –** It is a keyword which points to the **current passed instance**. But it **need not be passed every time while calling an instance method**.

|  |
| --- |
| # Python program to demonstrate instance methods  **class** shape:      # Calling Constructor  **def** \_\_init\_\_(self, edge, color):          self.edge **=** edge          self.color **=** color        # Instance Method  **def** finEdges(self):  **return** self.edge        # Instance Method  **def** modifyEdges(self, newedge):          self.edge **=** newedge    # Driver Code  circle **=** shape(0, 'red')  square **=** shape(4, 'blue')    # Calling Instance Method  **print**("No. of edges for circle: "**+** str(circle.finEdges()))    # Calling Instance Method  square.modifyEdges(6)  **print**("No. of edges for square: "**+** str(square.finEdges())) |

**Output**

No. of edges for circle: 0

No. of edges for square: 6

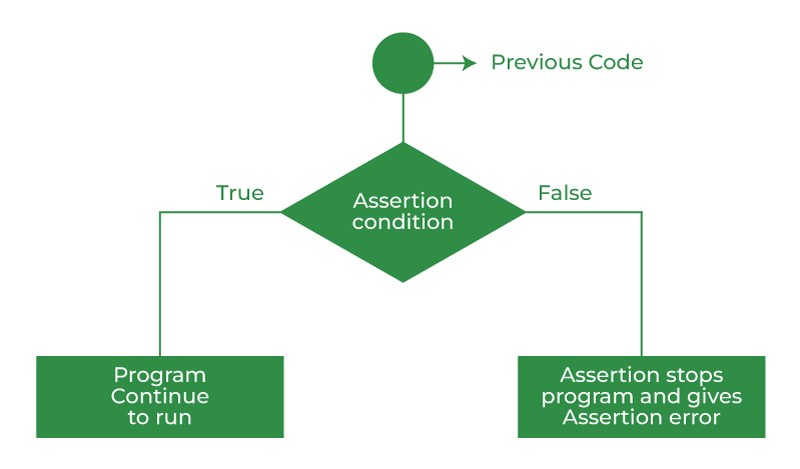
**ASSERT KEYWORD**

**Python Assertions** in any programming language are the **debugging tools that help in the smooth flow of code**. **Assertions are mainly assumptions that a programmer knows or always wants to be true and hence puts them in code so that failure of these doesn’t allow the code to execute further.**

## Assert Keyword 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 raises an AssertionError along with the optional message provided.**

### Flowchart of Python Assert Statement



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

**Parameters:**

* **condition :**The boolean condition returning true or false.
* **error\_message :**The optional argument to be printed in console in case of AssertionError.

**Returns:**Returns AssertionError, in case the condition evaluates to false along with the error message which when provided.

### Python assert keyword without error message

This code is trying to demonstrate the use of assert in Python by checking whether the value of b is 0 before performing a division operation. a is initialized to the value 4, and b is initialized to the value 0. The program prints the message “The value of a / b is: “.The assert statement checks whether b is not equal to 0. Since b is 0, the assert statement fails and raises an AssertionError.  
Since an exception is raised by the failed assert statement, the program terminates and does not continue to execute the print statement on the next line.

|  |
| --- |
| # initializing number  a **=** 4  b **=** 0  # using assert to check for 0  print("The value of a / b is : ")  **assert** b !**=** 0  print(a **/** b) |

**Output:**

The value of a / b is :

---------------------------------------------------------------------------

AssertionError Traceback (most recent call last)

Input In [19], in <cell line: 10>()

8 # using assert to check for 0

9 print("The value of a / b is : ")

---> 10 assert b != 0

11 print(a / b)

AssertionError:

### Python assert keyword with an error message

This code is trying to demonstrate the use of assert in Python by checking whether the value of b is 0 before performing a division operation. a is initialized to the value 4, and b is initialized to the value 0. The program prints the message “The value of a / b is: “.The assert statement checks whether b is not equal to 0. Since b is 0, the assert statement fails and raises an AssertionError with the message **“Zero Division Error”**.

Since an exception is raised by the failed assert statement, the program terminates and does not continue to execute the print statement on the next line.

|  |
| --- |
| # Python 3 code to demonstrate  # working of assert  # initializing number  a **=** 4  b **=** 0  # using assert to check for 0  **print**("The value of a / b is : ")  **assert** b !**=** 0, "Zero Division Error"  print(a **/** b) |

**Output:**

AssertionError: Zero Division Error

### Assert Inside a Function

The assert statement is used inside a function in this example to verify that a rectangle’s length and width are positive before computing its area. The assertion raises an AssertionError with the message “Length and width must be positive” if it is false. If the assertion is true, the function returns the rectangle’s area; if it is false, it exits with an error. To show how to utilize assert in various situations, the function is called twice, once with positive inputs and once with negative inputs.

|  |
| --- |
| # Function to calculate the area of a rectangle  **def** calculate\_rectangle\_area(length, width):      # Assertion to check that the length and width are positive  **assert** length > 0 **and** width > 0, "Length and width "**+** "must be positive"      # Calculation of the area      area **=** length **\*** width      # Return statement  **return** area  # Calling the function with positive inputs  area1 **=** calculate\_rectangle\_area(5, 6)  **print**("Area of rectangle with length 5 and width 6 is", area1)  # Calling the function with negative inputs  area2 **=** calculate\_rectangle\_area(**-**5, 6)  print("Area of rectangle with length -5 and width 6 is", area2) |

**Output:**

AssertionError: Length and width must be positive

### Assert with boolean Condition

In this example, the assert statement checks whether the boolean condition x < y is true. If the assertion fails, it raises an AssertionError. If the assertion passes, the program continues and prints the values of x and y.

|  |
| --- |
| # Initializing variables  x **=** 10  y **=** 20  # Asserting a boolean condition  **assert** x < y  # Printing the values of x and y  **print**("x =", x)  **print**("y =", y) |

**Output:**

x = 10

y = 20

### Assert Type of Variable in Python

In this example, the assert statements check whether the types of the variables a and b are str and int, respectively. If any of the assertions fail, it raises an AssertionError. If both assertions pass, the program continues and prints the values of a and b.

|  |
| --- |
| # Initializing variables  a **=** "hello"  b **=** 42  # Asserting the type of a variable  **assert** type(a) **==** str  **assert** type(b) **==** int  # Printing the values of a and b  print("a =", a)  print("b =", b) |

**Output:**

a = hello

b = 42

### Asserting dictionary values

In this example, the assert statements check whether the values associated with the keys “apple”, “banana”, and “cherry” in the dictionary my\_dict are 1, 2, and 3 respectively. If any of the assertions fail, it raises an AssertionError. If all assertions pass, the program continues and prints the contents of the dictionary.

|  |
| --- |
| # Initializing a dictionary  my\_dict **=** {"apple": 1, "banana": 2, "cherry": 3}  # Asserting the contents of the dictionary  **assert** my\_dict["apple"] **==** 1  **assert** my\_dict["banana"] **==** 2  **assert** my\_dict["cherry"] **==** 3  # Printing the dictionary  **print**("My dictionary contains the following key-value pairs:", my\_dict) |

**Output:**

My dictionary contains the following key-value pairs:

{'apple': 1, 'banana': 2, 'cherry': 3}

## **Practical Application**

This has a much greater utility in the Testing and Quality Assurance roles in any development domain. Different types of assertions are used depending on the application. Below is a simpler demonstration of a program that only allows only the batch with all hot food to be dispatched, else rejects the whole batch.

|  |
| --- |
| # Python code to demonstrate working of assert Application  # initializing list of foods temperatures  batch **=** [40, 26, 39, 30, 25, 21]  # initializing cut temperature  cut **=** 26  # using assert to check for temperature greater than cut  **for** i **in** batch:  **assert** i >**=** 26, "Batch is Rejected"  **print** (str(i) **+** " is O.K." ) |

**Output:**

40 is O.K.

26 is O.K.

39 is O.K.

30 is O.K.

**Runtime Exception:**

AssertionError: Batch is Rejected

## **Why Use Python Assert Statement?**

In Python, the assert statement is a potent debugging tool that can assist in identifying mistakes and ensuring that your code is operating as intended. Here are several justifications for using assert:

1. **Debugging: Assumptions** made by your code can be **verified** with the assert statement. You may **rapidly find mistakes and debug your program** by placing assert statements throughout your code.
2. **Documentation:**The use of assert statements in your code might **act as documentation**. Assert statements make it **simpler for others to understand and work with your code** since they explicitly describe the assumptions that your code is making.
3. **Testing:**In order to ensure that certain requirements are met, assert statements are frequently used in **unit testing**. You can make sure that your code is working properly and that any changes you make don’t damage current functionality by incorporating assert statements in your tests.
4. **Security:**You can use assert to **check that program inputs comply with requirements and validate them**. By doing so, security flaws like **buffer overflows and SQL injection attacks may be avoided**.

## **What are Namespaces in Python ?**

A python namespace is **a container where names are mapped to objects**, they are used to **avoid confusion in cases where the same names exist in different namespaces**. They are **created by modules, functions, classes**, etc. This helps to **keep the code organized** and **prevents naming conflicts between different modules**.

## **What is Scope in Python**

A scope defines the **hierarchical order in which the namespaces have to be searched in order to obtain the mappings of *name-to-object*(variables)**. It is a context in which variables exist and from which they are referenced. It defines the **accessibility and the lifetime of a variable**. Let us take a simple example as shown below:

|  |
| --- |
| pi **=** 'outer pi variable'    **def** print\_pi():      pi **=** 'inner pi variable'      print(pi)    print\_pi()  print(pi) |

**Output:**

inner pi variable

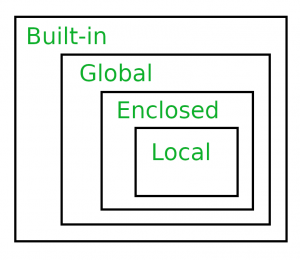
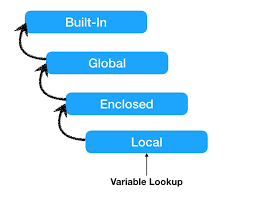
outer pi variable

The above program gives different outputs because the same variable name *pi* resides in different namespaces, one inside the function *print\_pi* and the other in the upper level. When *print\_pi()* gets executed, ‘*inner pi variable*‘ is printed as that is *pi* value inside the function namespace. The value ‘*outer pi variable*‘ is printed when *pi* is referenced in the outer namespace. From the above example, we can guess that there definitely is a **rule** which is followed, in order in deciding from which namespace a variable has to be picked.

## **Scope resolution LEGB rule In Python**

In Python, the **LEGB rule** is **used to decide the order in which the namespaces are to be searched for scope resolution**. The scopes are listed below in terms of **hierarchy(highest to lowest/narrowest to broadest)**:

* **Local(L):** Defined inside function/class
* **Enclosed(E):** Defined inside enclosing functions(Nested function concept)
* **Global(G):** Defined at the uppermost level
* **Built-in(B):** Reserved names in Python builtin modules

****

### ****Local Scope in Python****

Local scope refers to **variables defined in the current function**. **Always, a function will first look up a variable name in its local scope.** Only if it **does not find it there**, the **outer scopes are checked**.

|  |
| --- |
| # Local Scope  pi **=** 'global pi variable'  **def** inner():      pi **=** 'inner pi variable'      print(pi)    inner() |

**Output:**

inner pi variable

On running the above program, the execution of the *inner* function prints the value of its local(highest priority in LEGB rule) variable *pi* because it is defined and available in the local scope.

### ****Local and Global Scopes in Python****

If a variable is **not defined in the local scope**, then, it is **checked** for in the **higher scope**, in this case, the **global scope**.

|  |
| --- |
| # Global Scope  pi **=** 'global pi variable'  **def** inner():      pi **=** 'inner pi variable'      print(pi)    inner()  print(pi) |

**Output:**

inner pi variable

global pi variable

Therefore, as expected the program prints out the value in the local scope on the execution of *inner()*. It is because it is defined inside the function and that is the first place where the variable is looked up. The *pi* value in global scope is printed on the execution of *print(pi)* on line 9.

### ****Local, Enclosed, and Global Scopes in Python****

For the **enclosed scope**, we need to **define an outer function enclosing the inner function**, comment out the local *pi* variable of the inner function and refer to *pi* using **the *nonlocal* keyword**.

|  |
| --- |
| # Enclosed Scope  pi **=** 'global pi variable'    **def** outer():      pi **=** 'outer pi variable'  **def** inner():          # pi = 'inner pi variable'          nonlocal pi          print(pi)      inner()    outer()  print(pi) |

**Output:**

outer pi variable

global pi variable

When *outer()* is executed, *inner()* and consequently the *print* functions are executed, which print the value the enclosed *pi* variable. Since *pi* is referred with the ***nonlocal* keyword**, it means that ***pi* needs to be accessed from the *outer* function(i.e the outer scope)**. To summarize, **the *pi* variable is not found in local scope, so the higher scopes are looked up. It is found in both enclosed and global scopes. But as per the LEGB hierarchy, the enclosed scope variable is considered even though we have one defined in the global scope.**

### ****Local, Enclosed, Global, and Built-in Scopes****

The final check can be done by importing *pi* from *math* module and commenting on the global, enclosed, and local *pi* variables as shown below:

|  |
| --- |
| # Built-in Scope  **from** math **import** pi  # pi = 'global pi variable'    **def** outer():      # pi = 'outer pi variable'  **def** inner():          # pi = 'inner pi variable'          print(pi)      inner()  outer() |

**Output:**

3.141592653589793

Since, *pi* is not defined in either local, enclosed or global scope, the built-in scope is looked up i.e the *pi* value imported from the *math* module. Since the program is able to find the value of *pi* in the outermost scope, the following output is obtained.

**PYTHON CLOSURES**

**Nested functions in Python**

A **function that is defined inside another function** is known as a nested function. Nested functions are able to **access variables of the enclosing scope**.

**Definition**

Python closure is a **nested**[**function**](https://www.programiz.com/python-programming/function) that **allows** us to **access**[**variables**](https://www.programiz.com/python-programming/variables-constants-literals)**of the outer function** **even after the outer function is closed**.They are used in Decorators.

**When and Why to Use Closures**

1. As Python closures are **used as callback functions**, they provide some sort of **data hiding**.
2. When we have **few functions in our code**, closures in Python prove to be an efficient way. But if we need to have many functions, then go for class (OOP).
3. We may have variables in the global scope that are not used by many functions at times. Instead of defining variables in global scope, consider using a closure. This helps us to **reduce the use of global variables**.
4. A class in the Python programming language always has the **\_\_init\_\_ method**. If you only have **one extra method**, an elegant solution would be to **use a closure rather than a class**. Because this **improves code readability and even reduces the programmer’s workload**. Closures in Python can thus be used to **avoid the needless use of a class**.

|  |
| --- |
| def greet(name):  # inner function  def display\_name():  print("Hi", name)    # call inner function  display\_name()  # call outer function  greet("John")  # Output: Hi John |

|  |
| --- |
| # Python program to illustrate closures  **def** outerFunction(text):    **def** innerFunction():  **print**(text)        # Note we are returning function WITHOUT parenthesis  **return** innerFunction    **if** \_\_name\_\_ **==** '\_\_main\_\_':      myFunction **=** outerFunction('Hey!')      myFunction() |

**PYTHON ITERATORS**

An **Iterator** in Python is an object that is **used to iterate over iterable objects like lists, tuples, dicts, and sets**. The Python iterators object is **initialized** using the **iter()**method. It uses the **next()** method for iteration.

1. **\_\_iter\_\_():** The iter() method is called for the **initialization of an iterator**. This **returns an iterator object**.
2. **\_\_next\_\_():**The next method **returns the next value for the iterable**. When we use a **for loop** to traverse any iterable object, **internally it uses the iter() method to get an iterator object**, which **further uses the next() method to iterate over**. This method **raises a StopIteration to signal the end of the iteration**.

|  |
| --- |
| string **=** "GFG"  ch\_iterator **=** iter(string)  print(next(ch\_iterator))  **print**(next(ch\_iterator))  print(next(ch\_iterator)) |

**Output :**

G

F

G

|  |
| --- |
| tup **=** ('a', 'b', 'c', 'd', 'e')    # creating an iterator from the tuple  tup\_iter **=** iter(tup)    **print**("Inside loop:")  # iterating on each item of the iterator object  **for** index, item **in** enumerate(tup\_iter):      print(item)        # break outside loop after iterating on 3 elements  **if** index **==** 2:  **break**    # we can print the remaining items to be iterated using next()  # thus, the state was saved  print("Outside loop:")  **print**(next(tup\_iter))  **print**(next(tup\_iter)) |

**Output:**

Inside loop:

a

b

c

Outside loop:

d

e

**PYTHON GENERATORS**

A **Generator** in Python is a **function that returns an iterator using the Yield keyword**.The yield keyword is used to produce a value from the generator.

Generators use a yield statement in which the **state of the function is saved** from the last call and can be picked up or resumed the next time we call a generator function. Another great advantage of the generator over a list is that it **takes much less memory**.

**def function\_name():  
 yield statement**

## **Generator Object**

Python Generator functions **return a generator object that is iterable**, i.e., can be used as an Iterator. Generator objects are **used either by calling the next method of the generator object or using the generator object in a “for in” loop**.

**Example:**

In this example, we will create a simple generator function in Python to generate objects using the next() function.

|  |
| --- |
| # A Python program to demonstrate use of  # generator object with next()    # A generator function  **def** simpleGeneratorFun():  **yield** 1  **yield** 2  **yield** 3    # x is a generator object  x **=** simpleGeneratorFun()    # Iterating over the generator object using next    # In Python 3, \_\_next\_\_()  print(next(x))  **print**(next(x))  print(next(x)) |

**Output:**

1  
2  
3

**Example:**

In this example, we will create two generators for Fibonacci Numbers, first a simple generator and second generator using a [for loop](https://www.geeksforgeeks.org/python-for-loops/).

|  |
| --- |
| # A simple generator for Fibonacci Numbers  **def** fib(limit):      # Initialize first two Fibonacci Numbers      a, b **=** 0, 1      # One by one yield next Fibonacci Number  **while** a < limit:  **yield** a          a, b **=** b, a **+** b  # Create a generator object  x **=** fib(5)    # Iterating over the generator object using next  # In Python 3, \_\_next\_\_()  **print**(next(x))  **print**(next(x))  print(next(x))  **print**(next(x))  **print**(next(x))    # Iterating over the generator object using for  # in loop.  print("\nUsing for in loop")  **for** i **in** fib(5):      print(i) |

**Output:**

0  
1  
1  
2  
3  
  
Using for in loop  
0  
1  
1  
2  
3

**Difference between Generator function and Normal function –**

* Once the **function yields**, the function is **paused** and the **control is transferred to the caller**.
* When the function terminates, **StopIteration** is **raised automatically** on further calls.
* L**ocal variables and their states are remembered** between successive calls.
* The generator function contains **one or more yield statements instead of a return statement**.
* As the methods like **\_next\_()** and **\_iter\_()** are **implemented** **automatically**, we can **iterate** through the items **using next().**

**GENERATORS EXPRESSIONS**

In Python, a **generator expression** is a concise way to **create a generator object**.

It is **similar to a**[**list comprehension**](https://www.programiz.com/python-programming/list-comprehension), but instead of creating a [list](https://www.programiz.com/python-programming/list), it creates a generator object that can be iterated over to produce the values in the generator.

### Generator Expression Syntax

(expression for item in iterable)

|  |
| --- |
| # Python code to illustrate generator expression  generator **=** (num **\*\*** 2 **for** num **in** range(10))  **for** num **in** generator:      print(num) |

**Output :**

0  
1  
4  
9  
16  
25  
36  
49  
64  
81

|  |
| --- |
| List1 = [1,2,3,4,5,6,7]    # List Comprehension  z = [x\*\*3 **for** x in list1]    # Generator expression  a = (x\*\*3 **for** x in list1)  print(z)  print(list(a)) |

**Output:**

[1, 8, 27, 64, 125, 216, 343]

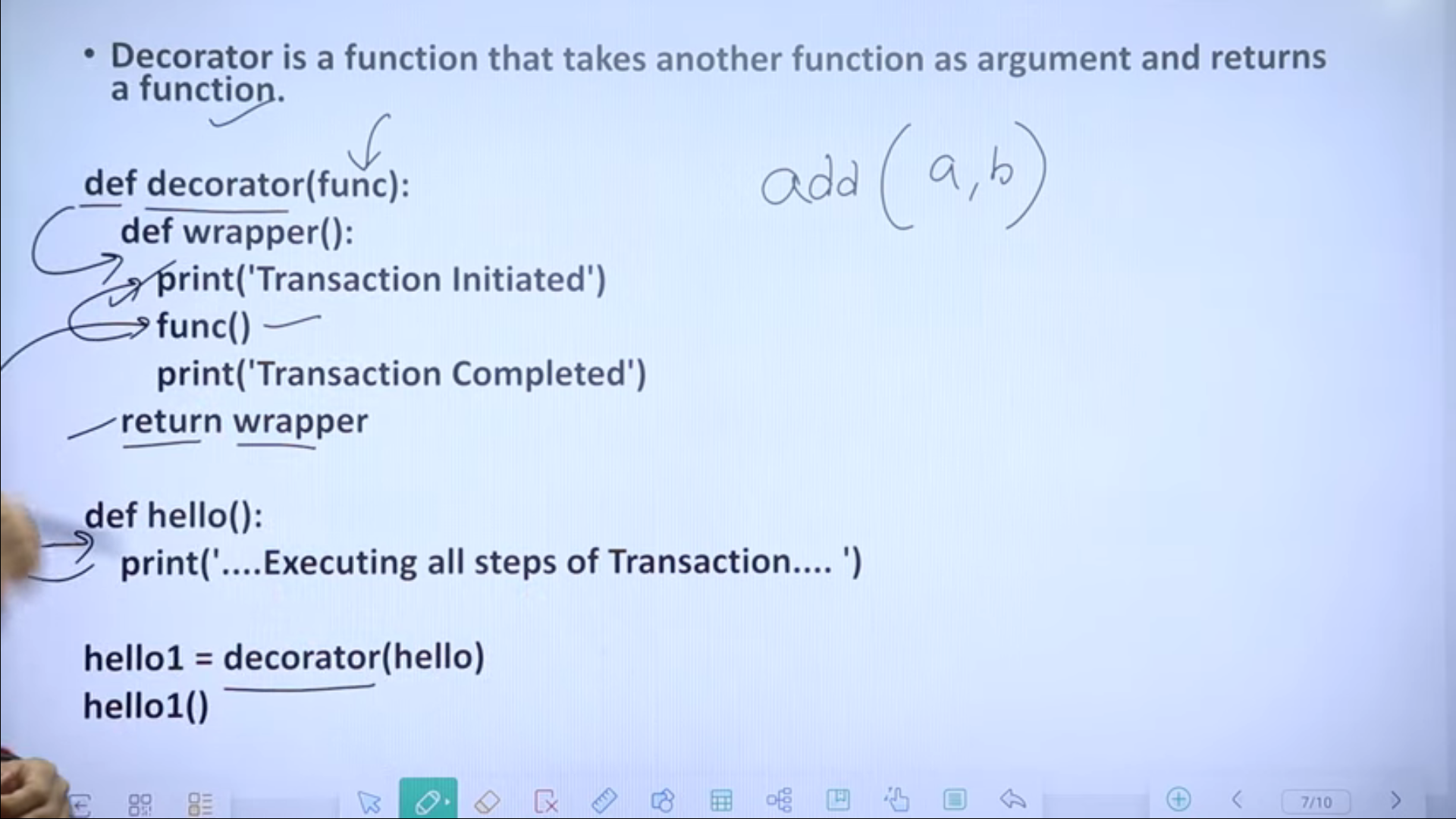
[1, 8, 27, 64, 125, 216, 343]

**DECORATORS IN PYTHON**

# **Decorators** are a very powerful and useful tool in Python since it allows programmers to **modify the behaviour of a function or class**. Decorators allow us to **wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it.**

### First Class Objects In Python, functions are first class objects which means that functions in Python can be used or passed as arguments. Properties of first class functions:

* A function is an **instance of the Object type**.
* You can **store the function in a variable**.
* You can **pass the function as a parameter to another function**.
* You can **return the function from a function**.
* You can **store them in data structures such as hash tables, lists**,etc.

****

