

UNIT-III

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

---

---

---


---

---

---

---

---



Learning Objectives

In this unit, we'll cover the following:

- Object Oriented Programming Concepts
- Classes and Objects
- Inheritance
- Polymorphism
- Abstract Classes
- Threads
- Exception Handling

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

---

---

---


---

---

---

---

---



Object Oriented Programming Concepts

- **Object-oriented programming (OOP)** is a method of structuring a program by bundling related properties and behaviors into individual **objects**.
- An object could represent a person with **properties** like a name, age, and address and **behaviors** such as walking, talking, breathing, and running. Or it could represent an email with properties like a recipient list, subject, and body and behaviors like adding attachments and sending.
- Object-oriented programming is an approach for modeling concrete, real-world things, like cars, as well as relations between things, like companies and employees, students and teachers, and so on. OOP models real-world entities as software objects that have some data associated with them and can perform certain functions.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

---

---

---


---

---

---

---

---



## OOPS in Python

- Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are very easy.
- Unlike procedure oriented programming, where the main emphasis is on functions, object oriented programming stresses on objects.
- The key takeaway is that objects are at the center of object-oriented programming in Python, not only representing the data, as in procedural programming, but in the overall structure of the program as well.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 4

---

---

---


---

---

---

---

---



## Overview of OOP Terminology

- **Class** – A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable** – A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member** – A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** – The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 5

---

---

---


---

---

---

---

---



## Overview of OOP Terminology continued...

- **Instance variable** – A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance** – The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** – An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation** – The creation of an instance of a class.
- **Method** – A special kind of function that is defined in a class definition.
- **Object** – A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading** – The assignment of more than one function to a particular operator.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 6

---

---

---


---

---

---

---

---



## Creating a Class in Python

Problem with primitive and advanced data types:  
Aarti=["Aarti Sharma",24, "Officer", 2018]

Supreet=["Supreet Kaur", "Sr. Manager", 2010]

- It can make larger code files more difficult to manage.
- It can introduce errors if not every employee has the same number of elements in the list.

A great way to make this type of code more manageable and more maintainable is to use **classes**.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

---

---

---


---

---

---

---

---



## Classes

- Classes are used to create user-defined data structures. Classes define functions called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data.
- A class is a blueprint for how something should be defined. It doesn't actually contain any data.
- While the class is the blueprint, an **instance** is an object that is built from a class and contains real data. Put another way, a class is like a form or questionnaire. An instance is like a form that has been filled out with information.

**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

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

---

---

---


---

---

---

---

---



## Defining a Class

All class definitions start with the **class** keyword, which is followed by the name of the class and a colon. Any code that is indented below the class definition is considered part of the class.

The first string inside the class is called docstring and has a brief description about the class. Although not mandatory, this is highly recommended.

```

Class My_Class:
    ' I am creating my first class in Python'
    x=5
  
```

A class creates a new local [namespace](#) where all its attributes are defined. Attributes may be data or functions.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

---

---

---


---

---

---

---

---



## Creating an Object in Python

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with *actual values*.

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.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 10

---

---

---


---

---

---

---

---



## Creating an Object in Python

### Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

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

```
p1 = My_Class()
print(p1.x)
```

The class object could be used to access different attributes. Attributes may be data or method. Methods of an object are corresponding functions of that class.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 11

---

---

---


---

---

---

---

---



## The \_\_init\_\_() Function

Class functions that begin with double underscore `_` are called special functions as they have special meaning.

Of one particular interest is the `__init__()` function. This special function gets called whenever a new object of that class is instantiated.

This type of function is also called constructors in Object Oriented Programming (OOP) languages like C++ and Java. We normally use it to initialize all the variables or object state.

Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation. It is run as soon as an object of a class is instantiated.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 12

---

---

---


---

---

---

---

---



## The `__init__()` Function

```

class Complex_Number:
    def __init__(self, r=0, i=0):
        self.real = r
        self.imag=i

    def get_data(self):
        print(f'{self.real} + {self.imag}i')
num1=Complex_Number(2,3)
num1.get_data()

num2=Complex_Number(5)
num2.attr = 10
print(num2.real, num2.imag, num2.attr)
print(num1.attr)

```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 13

---

---

---


---

---

---

---

---



## Object Methods

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

```

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

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

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

```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 14

---

---

---


---

---

---

---

---



## Self Parameter

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

- It does not have to be named **`self`**, you can call it whatever you like, but it has to be the first parameter of any function in the class
- 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.
- If we have a method that takes no arguments, then we still have to have one argument.
- 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.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 15

---

---

---


---

---

---

---

---



## Deleting Attributes and Objects

Any attribute of an object can be deleted anytime, using the ***del*** statement.

```
p1.age = 40
del p1.age
del p1.myfunc()
del p1
```

**del() method**  
A class can implement the special method `__del__()`, called a destructor, that is invoked when the instance is about to be destroyed. destroyed

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

---

---

---


---

---

---

---

---



## Deleting Attributes and Objects

```
class Point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
    def __del__(self):
        class_name = self.__class__.__name__
        print class_name, "destroyed"

pt1=point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3)
del pt1 del pt2 del pt3
This __del__() destructor prints the class name of an instance that is about to be
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

---

---

---


---

---

---

---

---



## Built-In Class Attributes

- Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute –
- **`__dict__`** – Dictionary containing the class's namespace.
- **`__doc__`** – Class documentation string or none, if undefined.
- **`__name__`** – Class name.
- **`__module__`** – Module name in which the class is defined. This attribute is "`__main__`" in interactive mode.
- **`__bases__`** – A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

---

---

---


---

---

---

---

---



### Built-In Class Attributes

```
class Employee:
    'Common base class for all employees'
    empCount=0

    def __init__(self, name, salary):
        self.name=name
        self.salary=salary
        Employee.empCount +=1

    def displayCount(self):
        print ("Total employee = ", Employee.empCount)

    def displayEmployee(self):
        print ("name = ", self.name, ", Salary = ", self.salary)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

19

---

---

---


---

---

---

---

---



### Built-In Class Attributes

```
print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

20

---

---

---


---

---

---

---

---



### Access Modifiers in Python : Public, Private and Protected

- Python uses ‘\_’ symbol to determine the access control for a specific data member or a member function of a class. Access specifiers in Python have an important role to play in securing data from unauthorized access and in preventing it from being exploited.
- A Class in Python has three types of access modifiers –
  1. Public Access Modifier
  2. Protected Access Modifier
  3. Private Access Modifier

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

21

---

---

---


---

---

---

---

---



## Public Access Modifier

The members of a class that are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default.

In previous examples :

```
self.name
self.age
myfunc()
```

Are all public members.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

---

---

---


---

---

---

---

---



## Protected Access Modifier

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared protected by adding a single underscore '\_' symbol before the data member of that class.

```
class student:
    _name = None
    _roll = None
    _branch = None
    def __init__(self, name, roll, branch):
        self._name = name
        self._roll = roll
        self._branch = branch
    def _displayRollAndBranch(self):
        print(self._roll)
        print(self._branch)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

---

---

---


---

---

---

---

---



## Protected Access Modifier

```
Class Derived_Stud(student):
    def __init__(self, name, roll, branch):
        student.__init__(self, name, roll, branch)

    def displayDetails:
        print("Name:", self.name)
        self._displayRollAndBranch()

Obj = Derived_Stud("Anuj", 170334, "Computer Applications")

Obj.displayDetails()
```

In the above program, `_name`, `_roll` and `_branch` are protected data members and `_displayRollAndBranch()` method is a protected method of the super class **Student**. The `displayDetails()` method is a public member function of the class **Derived\_Stud**.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

---

---

---

---


---

---

---

---





## Private Access Modifier

The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore '\_' symbol before the data member of that class.

```
class student:
    __name = None
    __roll = None
    __branch = None
    def __init__(self, name, roll, branch):
        self.__name = name
        self.__roll = roll
        self.__branch = branch
    def __displayDetails(self):
        print(self.__name)
        print(self.__roll)
        print(self.__branch)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 26

---

---

---


---

---

---

---

---



## Private Access Modifier

```
def accessPrivateMethod(self)
    self.__displayDetails()

Obj = student("Anuj", 170334, "Computer Applications")

Obj.accessPrivateMethod()
```

In the above program, `__name`, `__roll` and `__branch` are private data members and `__displayDetails()` method is a private member function of the class **student**. `accessPrivateMethod()` method is a public member function of the class **student** which can be accessed from anywhere within the program. The `accessPrivateMethod()` method accesses the private members of the class **student**.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 27

---

---

---


---

---

---

---

---



## Decorators in Python

- A **decorator** is a design pattern in **Python** that allows a user to add new functionality to an existing object without modifying its structure. Python has an interesting feature called **decorators** to add functionality to an existing code.
- This is also called **metaprogramming** because a part of the program tries to modify another part of the program at compile time.
- Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of the wrapped function, without permanently modifying it.
- Functions in Python can be used or passed as arguments.
  - 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.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 27

---

---

---


---

---

---

---

---



## Decorators in Python

**Example 1:** Treating the functions as objects.

```
def shout(text):
    return text.upper()
    print(shout("Hello"))

yell=shout
print(yell("Hello"))
```

**Example 2:** Passing the function as argument

```
def shout(text):
    return text.upper()
def whisper(text):
    return text.lower()
def greet(func):
    greeting = func("Hi ! I am created by a function passed as an argument")

    greet(shout)
    greet(whisper)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 28

---

---

---


---

---

---

---

---



## Decorators in Python

**Another example:**

```
def inc(x):
    return x+1

def dec(x):
    return x-1

def operator(func,x):
    result=func(x)
    return result

>>>operator(inc,21)
>>>operator(dec,21)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 29

---

---

---


---

---

---

---

---



## Decorators in Python

**Example 3:** Returning functions from another functions.

```
def is_called():
    def is_returned():
        print("Hello")
    return is_returned

new = is_called()

new()

Here is_returned() is a nested function which is defined and returned each time we call is_called().
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 30

---

---

---


---

---

---

---

---



## Decorators in Python

Functions and methods are called **callable** as they can be called.  
Basically, a decorator takes in a function, adds some functionality and returns it.

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner
def ordinary():
    print("I am ordinary")

>>>ordinary()
>>>pretty=make_pretty(ordinary)
>>>pretty()
```

The function ordinary() got decorated and the returned function was given the name pretty().  
We can see that the decorator function added some new functionality to the original function.  
The decorator acts as a wrapper.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 131

---

---

---


---

---

---

---

---



## Decorators in Python

Generally, we decorate a function and reassign it as,

```
ordinary = make_pretty(ordinary)
```

We can use the @symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example,

```
@make_pretty
def ordinary():
    print("I am ordinary")
```

It is equivalent to following:

```
def ordinary():
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 132

---

---

---


---

---

---

---

---



## Decorators with parameters

```
def smart_divide(func):
    def inner(a,b):
        print("I am going to divide ", a, " and ", b)
        if b==0:
            print("Can not divide")
            return
        return func(a,b)
    return inner

@smart_divide
def divide(a,b):
    print(a/b)

>>>divide(2,5)
>>>divide(2,0)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 133

---

---

---


---

---

---

---

---



## Different Types of Methods

There can be three types of methods in a class:

1. Instance methods
2. Class Methods
3. Static Methods

- The method **MyFunc()** is a regular instance method. That's the basic, no-frills method type you'll use most of the time. You can see the method takes one parameter, **self**, which points to an instance of class **Person** when the method is called (but of course instance methods can accept more than just one parameter).
- Through the **self** parameter, instance methods can freely access attributes and other methods on the same object. This gives them a lot of power when it comes to modifying an object's state.
- Not only can they modify object state, instance methods can also access the class itself through the **self.\_\_class\_\_** attribute. This means instance methods can also modify class state.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

34

---

---

---


---

---

---

---

---



## Class Methods

```
@classmethod
def classmethod(cls):
    return cls
```

Above method is marked with a **@classmethod** decorator to flag it as a *class method*.

Instead of accepting a **self** parameter, class methods take a **cls** parameter that points to the class—and not the object instance—when the method is called.

Because the class method only has access to this **cls** argument, it can't modify object instance state. That would require access to **self**. However, class methods can still modify class state that applies across all instances of the class.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

35

---

---

---


---

---

---

---

---



## Static Methods

```
@staticmethod
def staticmethod():
    return
```

Above method is marked with a **@staticmethod** decorator to flag it as a *static method*.

This type of method takes neither a **self** or a **cls** parameter (but of course it's free to accept an arbitrary number of other parameters).

Therefore a static method can neither modify object state nor class state. Static methods are restricted in what data they can access - and they're primarily a way to [namespace](#) your methods.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

36

---

---

---


---

---

---

---

---



## Class Vs Static Methods

- A class method takes cls as first parameter while a static method needs no specific parameters.
- A class method can access or modify class state while a static method can't access or modify it.
- In general, static methods know nothing about class state. They are utility type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as parameter.
- We use @classmethod decorator in python to create a class method and we use @staticmethod decorator to create a static method in python.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 37

---

---

---


---

---

---

---

---



## Class Vs Static Methods

**When to use what?**

- We generally use class method to create factory methods. Factory methods return class object ( similar to a constructor ) for different use cases.
- We generally use static methods to create utility functions.

**How to define a class method and a static method?**

To define a class method in python, we use @classmethod decorator and to define a static method we use @staticmethod decorator.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 38

---

---

---


---

---

---

---

---



## Class Vs Static Methods

```

from datetime import date
class person:
    def __init__(self, name, age):
        self.name=name
        self.age=age
    @classmethod
    def fromBirthYear(cls, name, year):
        return cls(name, date.today().year)

    @staticmethod
    def isAdult(age):
        return age>=18
person1= person("Anuj", 21)
person2= person.fromBirthYear("Anuj", 2000)
print(person1.age)
print(person2.age)

```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 39

---

---

---


---

---

---

---

---



## Data Hiding in Python

**Nothing in Python is truly private; internally,** the names of private methods and attributes are mangled and unmangled on the fly to make them seem inaccessible by their given names.

In Python, we use double underscore (Or `__`) before the attributes name and those attributes will not be directly visible outside.

```
class myClass:
    __hiddenVariable = 0
    def add(self, increment):
        self.__hiddenVariable += increment
        print(self.__hiddenVariable)

myObject = myClass()
myObject.add(2)
myObject.add(5)
print(myObject.__hiddenVariable)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 40

---

---

---


---

---

---

---

---



## Data Hiding in Python

In the previous code, we tried to access hidden variable outside the class using object and it threw an exception.

Now try,  
`Print(myObject.__myClass.__hiddenVariable)`

Python protects those members by internally changing the name to include the class name. You can access such attributes as `object.__className.__attrName`.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 41

---

---

---


---

---

---

---

---



## Inheritance in Python

Inheritance is the capability of one class to derive or inherit the properties from another class. The benefits of inheritance are:  
 It represents real-world relationships well.

- It provides **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.

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.

**Different forms of Inheritance:**

- Single inheritance:** When a child class inherits from only one parent class, it is called single inheritance.

```
class derived-class(base class):
    <class-suite>
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 42

---

---

---


---

---

---

---

---



## Inheritance in Python

**2. Multiple inheritance:** When a child class inherits from multiple parent classes, it is called multiple inheritance. Unlike Java and like C++, Python supports multiple inheritance. We specify all parent classes as a comma-separated list in the bracket.

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

**3. Multilevel inheritance:** Multi-level inheritance is achieved when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is achieved in python.

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 43

---

---

---


---

---

---

---

---



## Inheritance in Python continued ...

A simple example of inheritance in Python:

```
class person:
    def __init__(self, fname, lname):
        self.fname=fname
        self.lname=lname
    def printName(self):
        print(self.fname, self.lname)

x = person('Anuj', 'Rastogi')
x.printName()

class student(person):
    pass
y=student('Aarti','Sharma')
y.printName()
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 44

---

---

---


---

---

---

---

---



## Inheritance in Python continued ...

So far we have created a child class that inherits the properties and methods from its parent. Now, we want to add the `__init__()` function to the child class

```
class student(person):
    def __init__(self, lname,fname):
```

When you add the `__init__()` function to the child class, it will no longer inherit the parent's `__init__()` function. The child's `__init__()` function **overrides** the parent's `__init__()` function.

To keep the inheritance of the parent's `__init__()`, add a call to the parent's `__init__()`

```
class student(person):
    def __init__(self, lname,fname):
        person. __init__(self, lname,fname)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 45

---

---

---


---

---

---

---

---



## Inheritance in Python continued ...

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

```
class student(person):
    def __init__(self, lname, fname):
        super().__init__(lname, fname)
```

By using the ***super()*** function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

**Add Properties**

```
class student(person):
    def __init__(self, lname, fname):
        super().__init__(lname, fname)
        self.graduationyear = 2022
```

***graduationyear*** should be passed to ***\_\_init\_\_()*** function

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 46

---

---

---


---

---

---

---

---



## Inheritance in Python continued ...

**Add Methods**

```
class student(person):
    def __init__(self, lname, fname):
        super().__init__(lname, fname)
        self.graduationyear = 2022

    def welcome(self):
        print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

Two built-in functions ***isinstance()*** and ***issubclass()*** are used to check inheritances.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 47

---

---

---


---

---

---

---

---



## Inheritance in Python continued ...

**Method Overriding**

- We can provide some specific implementation of the parent class method in our child class.
- When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding.
- We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 48

---

---

---

---


---

---

---

---





## Inheritance in Python continued ...

```

class Bank:
    def getroi(self):
        return 10;
class SBI(Bank):
    def getroi(self):
        return 7;

class ICICI(Bank):
    def getroi(self):
        return 8;
b1 = Bank()
b2 = SBI()
b3 = ICICI()
print("Bank Rate of interest:", b1.getroi());
print("SBI Rate of interest:", b2.getroi());
print("ICICI Rate of interest:", b3.getroi());

```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 49

---

---

---


---

---

---

---

---



## Polymorphism in Python

The word polymorphism means having many forms. In programming, polymorphism means same function name (but different signatures) being used for different types.

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

**Polymorphism and Inheritance**

- The child classes in Python also inherit methods and attributes from the parent class. We can redefine certain methods and attributes specifically to fit the child class, which is known as **Method Overriding**.
- Polymorphism allows us to access these overridden methods and attributes that have the same name as the parent class.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 50

---

---

---


---

---

---

---

---



## Polymorphism in Python continued...

**Polymorphism with Class Methods**

```

class India:
    def capital(self):
        print('New Delhi')
    def language(self):
        print('Hindi, English and other regional languages')

class USA:
    def capital(self):
        print('Washington, D.C.')
    def language(self):
        print('English')

obj_ind=India()
obj_usa=USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()

```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 51

---

---

---


---

---

---

---

---



## Polymorphism in Python continued...

**Built-in polymorphic functions:**

```
print(len("BVICAM"))
print(len(["Python", "Java", "C++"]))
print(len("Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"))
```

**User defined polymorphic functions :**

```
def add(x,y,z=0):
    return x+y+z
print(add(2,3))
print(add(2,3,4))
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 62

---

---

---

---

---


---

---

---

---

---



## Operator Overloading

- Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class.
- The same built-in operator or function shows different behavior for objects of different classes, this is called *Operator Overloading*.

class point:

```
def __init__(self, x,y):
    self.x=x
    self.y=y

p1=point(2,3)
p2=point(5,8)

print(p1+p2)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 63

---

---

---

---

---


---

---

---

---

---



- We can change the meaning of an operator in Python depending upon the operands used.
- When we use an operator on user defined data types then automatically a special function or magic function associated with that operator is invoked. Changing the behavior of operator is as simple as changing the behavior of method or function.
- We define methods in your class and operators work according to that behavior defined in methods. When we use + operator, **the magic method \_\_add\_\_** is automatically invoked in which the operation for + operator is defined. There by changing this magic method's code, we can give extra meaning to the + operator.

```
def __add__(self, other):
    x = self.x + other.x
    y = self.y + other.y
    return point(x,y)
```

Now try

```
print(p1+p2)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 64

---

---

---

---

---


---

---

---

---

---



## Magic Functions for Operator Overloading

**Binary Operators:**

+	<code>__add__(self, other)</code>
-	<code>__sub__(self, other)</code>
*	<code>__mul__(self, other)</code>
/	<code>__truediv__(self, other)</code>
//	<code>__floordiv__(self, other)</code>
%	<code>__mod__(self, other)</code>
**	<code>__pow__(self, other)</code>
>>	<code>__rshift__(self, other)</code>
<<	<code>__lshift__(self, other)</code>
&	<code>__and__(self, other)</code>
	<code>__or__(self, other)</code>
^	<code>__xor__(self, other)</code>

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

65

---

---

---


---

---

---

---

---



## Magic Functions for Operator Overloading

**Comparison Operators :**

<	<code>__lt__(self, other)</code>
>	<code>__gt__(self, other)</code>
<=	<code>__le__(self, other)</code>
>=	<code>__ge__(self, other)</code>
==	<code>__eq__(self, other)</code>
!=	<code>__ne__(self, other)</code>

**Unary Operators :**

-	<code>__neg__(self, other)</code>
+	<code>__pos__(self, other)</code>
~	<code>__invert__(self, other)</code>

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

66

---

---

---


---

---

---

---

---



## Magic Functions for Operator Overloading

**Assignment Operators :**

-=	<code>__isub__(self, other)</code>
+=	<code>__iadd__(self, other)</code>
*=	<code>__imul__(self, other)</code>
/=	<code>__idiv__(self, other)</code>
//=	<code>__ifloordiv__(self, other)</code>
%=	<code>__imod__(self, other)</code>
**=	<code>__ipow__(self, other)</code>
>>=	<code>__irshift__(self, other)</code>
<<=	<code>__ilshift__(self, other)</code>
&=	<code>__iand__(self, other)</code>
=	<code>__ior__(self, other)</code>
^=	<code>__ixor__(self, other)</code>

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

67

---

---

---


---

---

---

---

---



## Abstract Classes

- A class is called an **Abstract class** if it contains **one or more abstract methods**. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and its abstract methods must be implemented by its subclasses.
- An abstract class can be considered as a blueprint for other classes.
- While we are designing large functional units we use an abstract class. When we want to provide a common interface for different implementations of a component, we use an abstract class.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 58

---

---

---


---

---

---

---

---



## How Abstract Base classes work

By default, Python does not provide abstract classes. Python comes with a module that provides the base for defining Abstract Base classes(ABC) and that module name is abc. **abc** works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base. A method becomes abstract when decorated with the keyword **@abstractmethod**. For Example –

```

import abc
class polygon(abc.ABC)
    @abc.abstractmethod
    def noofsides(self):
        pass
  
```

You may also provide class methods and static methods in abstract base class by decorators **@abstractclassmethod** and **@abstractstaticmethod** method decorators respectively.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 59

---

---

---


---

---

---

---

---



## Abstract Class continued...

```

class Triangle(polygon):
    def noofsides(self):
        print("I have 3 sides")

class Pentagon(polygon):
    def noofsides(self):
        print("I have 5 sides")

class Hexagon(polygon):
    def noofsides(self):
        print("I have 6 sides")

R = Triangle()
R.noofsides()
  
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 60

---

---

---


---

---

---

---

---



## Threads in Python

**Thread**

In computing, a **process** is an instance of a computer program that is being executed. Any process has 3 basic components:

- An executable program.
- The associated data needed by the program (variables, work space, buffers, etc.)
- The execution context of the program (State of process)

A **thread** is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System).

In simple words, a **thread** is a sequence of such instructions within a program that can be executed independently of other code. For simplicity, you can assume that a thread is simply a subset of a process!

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

61

---

---

---


---

---

---

---

---



## Threads continued...

A thread contains all this information in a **Thread Control Block (TCB)**:

- **Thread Identifier:** Unique id (TID) is assigned to every new thread
- **Stack pointer:** Points to thread's stack in the process. Stack contains the local variables under thread's scope.
- **Program counter:** a register which stores the address of the instruction currently being executed by thread.
- **Thread state:** can be running, ready, waiting, start or done.
- **Thread's register set:** registers assigned to thread for computations.
- **Parent process Pointer:** A pointer to the Process control block (PCB) of the process that the thread lives on.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

62

---

---

---


---

---

---

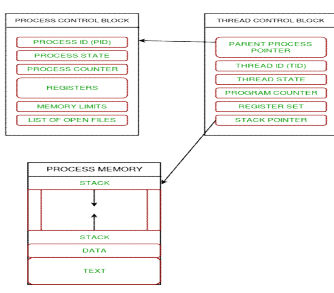
---

---



## Threads continued...

Consider the diagram below to understand the relation between process and its thread:



© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

63

---

---

---


---

---

---

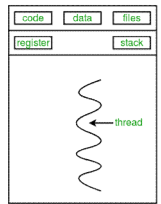
---

---

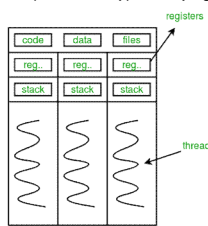


## Multithreading

- Multiple threads can exist within one process where:
- Each thread contains its own **register set** and **local variables (stored in stack)**.
- All thread of a process share **global variables (stored in heap)** and the **program code**.



single-threaded process



multithreaded process

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 64

---

---

---

---

---


---

---

---

---

---



## Multithreading continued...

**Multithreading** is defined as the ability of a processor to execute multiple threads concurrently.

Running several threads is similar to running several different programs concurrently, but with the following benefits –

- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
- Threads sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.

- It can be pre-empted (interrupted)
- It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 65

---

---

---

---

---


---

---

---

---

---



## Multithreading continued...

There are two ways of accessing Python threads. These are by using:

- Thread Module  
`t = thread.start_new_thread(function, args [, kwargs] )`
- Threading module  
`t = threading.Thread(function, args [, kwargs] )`

It is to be noted that the 'thread' module has been considered as of lesser use, and hence users get to use the 'threading' module instead. Another thing has to keep in mind that the module 'thread' treats the thread as a function, whereas the 'threading' is implemented as an object.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 66

---

---

---

---

---


---

---

---

---

---



### Multithreading continued...

```
import threading
def print_cube(num):
    print('Cube = ', num * num * num)
def print_square(num):
    print('Square = ', num * num )

if __name__ == "__main__" :
    t1 = threading.Thread(target=print_square, args=(10,))
    t2 = threading.Thread(target=print_cube, args=(10,))

    t1.start()
    t2.start()

    t1.join()
    t2.join()
    print("Done!")
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 67

---

---

---


---

---

---

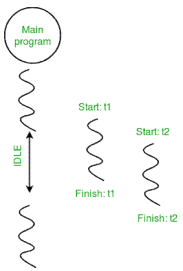
---

---



### Multithreading continued...

Consider the diagram below for a better understanding of how this program works:



© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 68

---

---

---


---

---

---

---

---



### Another Example of Multithreading

```
import threading
import os
def task1():
    print("Task 1 assigned to thread: ", threading.current_thread().name)
    print("ID of process running Task 1", os.getpid())
def task2():
    print("Task 2 assigned to thread: ", threading.current_thread().name)
    print("ID of process running Task 2", os.getpid())
if __name__ == "__main__" :
    print("ID of process running MAIN program: ", os.getpid())
    print("Main thread name : ", threading.current_thread().name)
    t1= threading.Thread(target=task1, name='t1')
    t2=threading.Thread(target=task2,name='t2')
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 69

---

---

---


---

---

---

---

---



## Python Program for Various Thread States

- There are five thread states - **new, runnable, running, waiting and dead**.
- Among these five Of these five, we will majorly focus on three states - running, waiting and dead.
- A thread gets its resources in the running state, waits for the resources in the waiting state; the final release of the resource, if executing and acquired is in the dead state.
- The following Python program with the help of start(), sleep() and join() methods will show how a thread entered in running, waiting and dead state respectively.

**Step 1** – Import the necessary modules, for example : <threading> and <time>

**Step 2** – Define a function, which will be called while creating a thread.

**Step 3** – We are using the sleep() method of time module to make our thread waiting for say 2 seconds.

**Step 4** – Now, we are creating a thread (e.g T1), which takes the argument of the function defined above.

**Step 5** – Now, with the help of the start() function we can start our thread. It will produce the message, which has been set by us while defining the function.

**Step 6** – Now, at last we can kill the thread with the join() method after it finishes its execution.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 70

---

---

---

---

---


---

---

---

---

---



## Daemon Thread in Python

- A **daemon thread** is a thread that dies whenever the main thread dies, it is also called a non-blocking thread.
- Usually, the main thread should wait for other threads to finish in order to quit the program, but if you set the daemon flag, you can let the thread do its work and forget about it, and when the program quits, it will be killed automatically.
- For example, you may want to make a thread that watches for log files in your program, and alert you when a critical error is occurred.
- Usually our main program implicitly waits until all other threads have completed their work.
- The default setting for a thread is non-daemon. To designate a thread as a daemon, we call its **setDaemon()** method with a boolean argument.

Example: test-Daemon.py

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 71

---

---

---

---

---


---

---

---

---

---



## Join() method in Python

On invoking the **join()** method, the calling thread gets blocked until the thread object (on which the thread is called) gets terminated. The thread objects can terminate under any one of the following conditions:

- Either normally.
- Through an ill-handled exception.
- Till the optional timeout occurs.

Hence the join() method indicates wait till the thread terminates. We can also specify a timeout value to the join() method. In such a situation the calling thread may ask the thread to stop by sending a signal through an event object. The join() method can be called multiple times.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 72

---

---

---

---

---

---


---

---

---

---





## threading.enumerate()

**threading.enumerate()** returns a list of all Thread objects currently alive. The list includes daemon threads, and the main thread. It excludes terminated threads and threads that have not yet been started.

```
for t in threading.enumerate():
    if t is main_thread:
        continue
    logging.debug('Name of the thread = %s', t.getName())
```

For example : test-Enumerate.py

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 73

---

---

---


---

---

---

---

---



## Thread Synchronization in Python

Thread synchronization is defined as a mechanism which ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as **critical section**.

Following issues may arise while implementing concurrent programming or applying synchronizing primitives:

1. Dead Lock
2. Race condition

In Python, we can implement synchronization by using the following concepts

- **Lock**
- **RLock**
- **Semaphore**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 74

---

---

---


---

---

---

---

---



## Synchronization By using Locks in python:

Locks are the most fundamental synchronization mechanism provided by the threading module. We can create Lock object as follows,

```
l=Lock()
```

The Lock object can be held by only one thread at a time. If any other thread wants the same lock then it will have to wait until the other one releases it. It's similar to waiting in line to book a train ticket, public telephone booth etc.

- **acquire() method:** A Thread can acquire the lock by using acquire() method.  
`l.acquire()`
- **release() method:** A Thread can release the lock by using release() method.  
`l.release()`

Examples : Test-Lock.py and Test-Lock1.py

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 75

---

---

---


---

---

---

---

---



### Problem with Simple Lock in Python:

- The standard lock object does not care which thread is currently holding that lock. If the lock is being held by one thread, and if any other thread tries to acquire the lock, then it will be blocked, even if it's the same thread that is already holding the lock.
- So, if the Thread calls recursive functions or nested access to resources, then the thread may try to acquire the same lock again and again, which may result in blocking of our thread. Hence Traditional Locking mechanism won't work for executing recursive functions.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 176

---

---

---


---

---

---

---

---



### Synchronization By using RLock concept in Python:

- To overcome the above problem of Simple Lock, we should go for RLock(Reentrant Lock). Reentrant means the thread can acquire the same lock again and again. This will block the thread only if the lock is held by any other thread. Reentrant facility is available only for owner thread but not for other threads.
- This RLock keeps track of recursion level and hence for every acquire() there should be a release() call available.
- The number of acquire() calls and release() calls should be matched then for the lock to be released i.e if there are two acquire calls then there should be two release calls for the lock to be released. If there is only one release call for two acquire calls then the lock wont be released.

Example : test-RLock()

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 177

---

---

---


---

---

---

---

---



### Difference between Lock and RLock in Python:

Lock	RLock
A Lock object can be acquired by only one thread at a time. Even the owner thread also cannot acquire multiple times.	A RLock object can be acquired by only one thread at a time, but the owner thread can acquire the same lock object multiple times.
Not suitable to execute recursive functions and nested access calls	Best suitable to execute recursive functions and nested access calls
In this case the Lock object will only see whether its Locked or unlocked and it will never hold or take care of owner thread and recursion level.	In this case RLock object will see whether its Locked or unlocked and also about owner thread information and recursion level.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 178

---

---

---


---

---

---

---

---



## Synchronization by using Semaphore in Python:

- Semaphore provides threads with synchronized access to a limited number of resources.
- A semaphore is just a variable. The variable reflects the number of currently available resources. For example, a parking lot with a display of number of available slots on a specific level of a shopping mall is a semaphore.
- The value of semaphore cannot go less than zero and greater than the total number of the available resources.
- The semaphore is associated with two operations – **acquire** and **release**.
- When one of the resources synchronized by a semaphore is "**acquired**" by a thread, the value of the semaphore is decremented.
- When one of the resources synchronized by a semaphore is "**released**" by a thread the value of the semaphore is incremented.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 79

---

---

---


---

---

---

---

---



## Semaphores in Python:

- The Semaphore class of the Python threading module implements the concept of semaphore.
- It has a constructor and two methods `acquire()` and `release()`.
- The `acquire()` method decreases the semaphore count if the count is greater than zero. Else it blocks till the count is greater than zero.
- The `release()` method increases the semaphore count and wakes up one of the threads waiting on the semaphore.

**Way to create an object of Semaphore :**

- `object_name.Semaphore()`  
In this case, by default value of the count variable is 1 due to which only one thread is allowed to access. It is exactly the same as the **Lock** concept.
- `object_name.Semaphore(n)`  
In this case, a Semaphore object can be accessed by n Threads at a time. The remaining Threads have to wait until releasing the semaphore.

Example : test-Semaphores.py

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 80

---

---

---


---

---

---

---

---



## Bounded Semaphore in Python:

Try this :

```
from threading import *
s=Semaphore(2)
s.acquire()
s.acquire()
s.release()
s.release()
s.release()
s.release()
s.release()
print("End")
```

In **Normal Semaphore**, discussed above, the release method can be called any number of times to increase the counter, irrespective of the acquire method. Sometimes, the number of `release()` calls can exceed

It is valid because in normal semaphore we can call `release()` any number of times. This may result in programming errors or may raise confusions. So, it is always recommended to use **Bounded Semaphore** which raises an error if the number of `release()` calls exceeds the number of `acquire()` calls. the number of `acquire()` calls also.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 81

---

---

---


---

---

---

---

---



## Bounded Semaphore in Python:

**Now try this:**

```
from threading import *
s=BoundedSemaphore(2)
s.acquire()
s.acquire()
s.release()
s.release()
s.release()
s.release()
print("End")
```

Example: test-semaphores2.py

The major idea of synchronization is to overcome data inconsistency problems. But the disadvantage of synchronization is it increases waiting time of threads and creates performance problems. Hence it is recommended to use synchronization only if the requirement demands.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 62

---

---

---

---

---


---

---

---

---

---



## Thread Synchronization using Event Object

The Event class object provides a simple mechanism which is used for communication between threads where one thread signals an event while the other threads wait for it.

So, when one thread which is intended to produce the signal produces it, then the waiting thread gets activated.

An internal flag is used by the event object known as the **event flag** which can be set as true using the set() method and it can be reset to false using the clear() method

The wait() method blocks a thread until the event flag for which it is waiting is set true by any other thread.

There are few useful functions used along with an event object:

1. isSet()
2. set()
3. clear()
4. wait([Timeout])

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 63

---

---

---

---

---

---

---

---

---

---



## Thread Synchronization using Event Object

**isSet Method:**  
This method returns true if and only if the internal flag is true.

**set() Method:**  
When this method is called for any event object then the internal flag is set to true. And as soon as set() method is called for any event all threads waiting for it are awakened.

**clear() Method:**  
This method resets the internal flag to false. Subsequently, threads calling wait() on the event for which clear() is called, it will block until the internal flag is not true again.

**wait([Timeout]) Method:**

- When we have to make any thread wait for an event, we can do so by calling this method on that event which has the internal flag set to false, doing so blocks the thread until the internal flag is true for the event.
- If the internal flag is true on entry, then the thread will never get blocked. Otherwise, it is blocked until another thread calls set() to set the flag to true, or until the optional timeout occurs. The timeout argument specifies a timeout for the operation in seconds.

Example : test-event.py

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 64

---

---

---

---

---


---

---

---

---

---



## Exception Handling

Error in Python can be of two types i.e. Syntax errors and Exceptions. Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

**Exceptions:** Exceptions are raised when the program is syntactically correct but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

```
marks =1000
a=marks / 0
print(a)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 85

---

---

---


---

---

---

---

---



## Exception Handling

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them –

- **Exception Handling**
- **Assertions**

Python provides a way to handle the exception so that the code can be executed without any interruption. If we do not handle the exception, the interpreter doesn't execute all the code that exists after the exception.

Python has many **built-in exceptions** that enable our program to run without interruption and give the output.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 86

---

---

---


---

---

---

---

---



## Exception Handling

**Common Exceptions**

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. A list of common exceptions that can be thrown from a standard Python program is given below.

- **ZeroDivisionError:** Occurs when a number is divided by zero.
- **NameError:** It occurs when a name is not found. It may be local or global.
- **IndentationError:** If incorrect indentation is given.
- **IOError:** It occurs when Input Output operation fails.
- **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.
- **ArithmeticError:** Base class for all errors that occur for numeric calculation.
- **OverflowError:** Raised when a calculation exceeds maximum limit for a numeric type.
- **IndexError:** Raised when an index is not found in a sequence.
- **SyntaxError:** Raised when there is an error in Python syntax.
- **ValueError:** Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 87

---

---

---


---

---

---

---

---



## Exception Handling

### Handling an exception

- If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try**: block. After the try: block, include an **except**: statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax

Here is simple syntax of *try...except...else* blocks –

```
try:  
    You do your operations here;  
    .....  
except ExceptionI:  
    If there is ExceptionI, then execute this block.  
except ExceptionII:  
    If there is ExceptionII, then execute this block.  
    .....  
else:  
    If there is no exception then execute this block.
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 88

---

---

---


---

---

---

---

---



## Exception Handling

Here are few important points about the syntax –

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

```
try:  
    fh = open("testfile", "w")  
    fh.write("This is my test file for exception handling!!")  
except IOError:  
    print "Error: can't find file or read data"  
else:  
    print "Written content in the file successfully"  
    fh.close()
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 89

---

---

---


---

---

---

---

---



## The *except* Clause with No Exceptions

You can also use the except statement with no exceptions defined as follows –

```
try:  
    You do your operations here;  
    .....  
except:  
    If there is any exception, then execute this block.  
    .....  
else:  
    If there is no exception then execute this block.
```

This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 90

---

---

---


---

---

---

---

---



## The *except* Clause with Multiple Exceptions

```

try:
    You do your operations here;
    .....
except(<Exception 1>, <Exception 2>, <Exception 3>):
    Block of statements
    .....
else:
    If there is no exception then execute this block.
  
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 91

---

---

---


---

---

---

---

---



## The try-finally Clause

Python provides the optional **finally** statement, which is used with the **try** statement. It is executed no matter what exception occurs and used to release the external resource. The finally block provides a guarantee of the execution.

```

try:
    fileptr = open("file2.txt","r")
    try:
        fileptr.write("Hi I am good")
    finally:
        fileptr.close()
        print("file closed")
except:
    print("Error")
  
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 92

---

---

---


---

---

---

---

---



## Raising exceptions

- An exception can be raised forcefully by using the **raise** clause in Python. It is useful in that scenario where we need to raise an exception to stop the execution of the program.
- For example, there is a program that requires 2GB memory for execution, and if the program tries to occupy 2GB of memory, then we can raise an exception to stop the execution of the program.
- The syntax to use the raise statement is given below.

```
Raise Exception_class, <value>
```

**Points to remember**

- To raise an exception, the raise statement is used. The exception class name follows it.
- An exception can be provided with a value that can be given in the parenthesis.
- To access the value **"as"** keyword is used. **"e"** is used as a reference variable which stores the value of the exception.
- We can pass the value to an exception to specify the exception type.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 93

---

---

---


---

---

---

---

---



## Raising exceptions

```
try:
    age = int(input("Enter the age:"))
    if(age<18):
        raise ValueError
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")

Raise the exception with message
try:
    num = int(input("Enter a positive integer: "))
    if(num <= 0):
        raise ValueError("That is a negative number!")
except ValueError as e:
    print(e)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

94

---

---

---


---

---

---

---

---



## Custom Exception

The Python allows us to create our exceptions that can be raised from the program and caught using the except clause. However, we suggest you read this section after visiting the Python object and classes.

```
class ErrorInCode(Exception):
    def __init__(self, data):
        self.data = data
    def __str__(self):
        return repr(self.data)

try:
    raise ErrorInCode(2000)
except ErrorInCode as ae:
    print("Received error:", ae.data)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

95

---

---

---


---

---

---

---

---



## Assertions in Python

- An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.
- 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**.
- Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.
- Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

**The assert Statement**

- When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an **AssertionError** exception.
- The **syntax** for assert is –  
assert Expression[, Argument]

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1

96

---

---

---

---


---

---

---

---





## Assertions in Python

Here is a function that converts a temperature from degrees Kelvin to degrees Fahrenheit. Since zero degrees Kelvin is as cold as it gets, the function bails out if it sees a negative temperature –

```
def KelvinToFahrenheit(Temperature):
    assert (Temperature >= 0), "Colder than absolute zero!"
    return ((Temperature-273)*1.8)+32

print KelvinToFahrenheit(273)
print int(KelvinToFahrenheit(505.78))
print KelvinToFahrenheit(-5)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 87

---

---

---


---

---

---

---

---



## Exception Class Hierarchy in Python:

- All exception classes are derived from the BaseException class.
- The BaseException is the base class of all other exceptions. User defined classes cannot be directly derived from this class, to derive user defined class, we need to use Exception class.

[Python Exception Hierarchy.docx](#)

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 88

---

---

---


---

---

---

---

---



## Composition in Python

- **Composition** is an object oriented design concept that models a **has a** relationship. In composition, a class known as **composite** contains an object of another class known to as **component**. In other words, a composite class **has a** component of another class.
- Composition allows composite classes to reuse the implementation of the components it contains. The composite class doesn't inherit the component class interface, but it can leverage its implementation.
- The composition relation between two classes is considered loosely coupled. That means that changes to the component class rarely affect the composite class, and changes to the composite class never affect the component class.
- This provides better adaptability to change and allows applications to introduce new requirements without affecting existing code.
- When looking at two competing software designs, one based on inheritance and another based on composition, the composition solution usually is the most flexible.
- Example : test-Composition.py

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 89

---

---

---


---

---

---

---

---



## Iterators in Python

A process that is repeated more than one time by applying the same logic is called an iteration. In programming languages like python, a loop is created with few conditions to perform iteration till it exceeds the limit. If the loop is executed 6 times continuously, then we could say the particular block has iterated 6 times.

```
a = [0, 5, 10, 15, 20]
for i in a:
    if i % 2 == 0:
        print(str(i)+' is an Even Number')
    else:
        print(str(i)+' is an Odd Number')
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 100

---

---

---


---

---

---

---

---



## Iterators in Python

- An iterator is an object which contains a countable number of values and it is used to iterate over iterable objects like list, tuples, sets, etc.
- Iterators are implemented using a class and a local variable for iterating is not required here. It follows lazy evaluation where the evaluation of the expression will be on hold and stored in the memory until the item is called specifically which helps us to avoid repeated evaluation.
- As lazy evaluation is implemented, it requires only 1 memory location to process the value and when we are using a large dataset then, wastage of RAM space will be reduced the need to load the entire dataset at the same time will not be there.

**How to use an iterator-**

- iter()** keyword is used to create an iterator containing an iterable object.
- next()** keyword is used to call the next element in the iterable object.
- After the iterable object is completed, to use them again reassign them to the same object.

```
iter_list = iter(['Java', 'Python', 'C++'])
print(next(iter_list))
print(next(iter_list))
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 101

---

---

---


---

---

---

---

---



## Generators in Python

- It is another way of creating iterators in a simple way where it uses the keyword "yield" instead of returning it in a defined function.
- Generators are implemented using a function.
- Just as iterators, generators also follow lazy evaluation.
- The yield function returns the data without affecting or exiting the function. It will return a sequence of data in an iterable format where we need to iterate over the sequence to use the data as they won't store the entire sequence in the memory.

Example : test-Generator.py

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1 102

---

---

---

---

---

---

---

---