CHAPTER 3

CLASSES AND PACKAGES

Chapter Outline

3.1 CLASS WITH ITS MEMBERS AND CREATE INSTANCES OF CLASS

Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy. However, here is small introduction of Object-Oriented Programming (OOP) to bring you at speed:

- **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.
- **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.
- **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.
- **Instance variable :** A variable that is defined inside a method and belongs only to the current instance of a class.
- **Data members :** 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.
- **Operator overloading :** The assignment of more than one function to a particular operator.
- **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.

CLASSES AND OBJECTS

- Classes and objects are the two main aspects of object oriented programming. In fact, a class is the basic building block in Python.
- A class creates a new type and object is an instance of the class.
- Classes provides a blue print or a template using which objects are created. In fact, in Python everything is an object or an instance of some class.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

CREATION OF CLASS :

A class is created with the keyword *class* and then writing the classname. The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    <statement-2>
    .
    .
    <statement-N>
```

Example :

```
class Student:
    def __init__(self):
        self.name="hari"
        self.branch="CSE"
    def display(self):
        print(self.name)
        print(self.branch)
```

Class definition starts with a keyword class followed by the class_name and a colon (:). The statement in the definition can be any of these - sequential instructions, decision control statements, loop statements and can even include function definitions.

Variables defined in a class are called ***class variables*** and functions defined inside a class are called ***class methods.*** Class variables and class methods are together known as class members. The class members can be accessed through class objects.

CREATING OBJECTS

- Once a class is defined, the next job is to create an object or instance of that class.
- The object can then access class variables and class methods using the dot operator (.).

Syntax :

```
Object_name = class_name ()
```

- The syntax for accessing a class member through the calss object is

```
Object_name.class_member_name
```

Example :

```
class A :
        rollno = 501
Aobj = A ( )
print ("Roll Number is " , Aobj.roll no)
```

Output :

```
Roll Number is 501
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Here is an example of a simple class definition:

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

    def say_hello(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

In this example, we define a class called Person that has two attributes (name and age) and one method (say_hello). The **__init__** method is a special method that is called when a new instance of the class is created. It takes two arguments (name and age) and initializes the name and age attributes with those values.

The say_hello method is a simple method that prints a greeting with the person's name and age.

To create an instance of the Person class, you use the class name followed by parentheses, like this:

```
person1 = Person("Suresh", 33)
```

This creates a new instance of the Person class with the name "Suresh" and age 33, and assigns it to the variable person1.

You can access the attributes of the instance using the dot notation, like this:

```
print(person1.name)  # Output: "Suresh"
print(person1.age)   # Output: 33
```

You can also call the methods of the instance using the dot notation, like this:

```
person1.say_hello()  # Output: "Hello, my name is Suresh and I am 33 years old."
```

You can create as many instances of the Person class as you want, each with its own name and age:

```
person2 = Person("Venkatesh", 29)
person3 = Person("Anvith", 5)
```

Now you have three instances of the Person class (person1, person2, and person3), each with its own name and age. You can call the say_hello method on each instance to get a personalized greeting.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

CONSTRUCTOR

A **constructor** is a special method that is used to initialize the instance variables of a class. In the constructor, we create the instance variables and initialize them with some starting values. The first parameter of the constructor will be 'self' variable that contains the memory address of the instance.

Creating a Constructor

- A constructor is a class function that begins with double underscore (_). The name of the constructor is always the same **__init__()**.
- When we create a class without a constructor, Python automatically creates a default constructor that doesn't do anything.
- In Python, Constructors can be parameterized and non-parameterized as well. The parameterized constructors are used to set custom value for instance variables that can be used further in the application.

A constructor can be defined in the following manner

```
def __init__( self ):
    self.name = "hari"
    self.branch = "CSE"
```

Here, the constructor has only one parameter, i.e. 'self' using 'self.name' and 'self.branch', we can access the instance variables of the class. A constructor is called at the time of creating an instance. So, the above constructor will be called when we create an instance as:

```
s1 = Student()
```

Let's take another example, we can write a constructor with some parameters in addition to 'self' as:

```
def __init__( self , n = ' ' , b = ' ' ):
    self.name = n
    self.branch = b
```

Here, the formal arguments are n and b whose default values are given as ''None" and ''None". Hence, if we do not pass any values to constructor at the time of creating an instance, the default values of those formal arguments are stored into name and branch variables. For example,

```
s1 = Student( )
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Since we are not passing any values to the instance, None and None are stored into name and branch. Suppose, we can create an instance as:

```
s1 = Student("Suresh", "CSE")
```

In this case, we are passing two actual arguments: "Suresh" and "CSE" to the Student instance.

Example :

```
class Student:
    def __init__(self, n='',b=''):
        self.name=n
        self.branch=b
    def display(self):
        print("Name: ", self.name)
        print("Branch: ", self.branch)
print("This is non parametrized constructor")
s1=Student() # Constructor - non parameterized
s1.display()
print("_____")
print("This is parametrized constructor")
s2=Student("Suresh", "CSE") # Constructor - parameterized
s2.display()
print("_____")
```

Output :

```
This is non parameterized constructor
Name:
Branch:
_____
This is parameterized constructor
Name: Suresh
Branch: CSE
_____
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

TYPES OF VARIABLES

The variables which are written inside a class are of two types:

(a) Instance Variables

(b) Class Variables or Static Variables

(a) **Instance Variables :** Instance variables are the variables whose separate copy is created in every instance. For example, if x is an instance variable and if we create 3 instances, there will be 3 copies of x in these 3 instances. When we modify the copy of x in any instance, it will not modify the other two copies.

EXAMPLE :

A Python program to understand instance variables.

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

    def print_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

# create persons
person1 = Person("Suresh", 33)
person2 = Person("Venkat", 29)

# print information of all persons
person1.print_info()
person2.print_info()

# modify information of one person
person1.name = "Raju"
person1.age = 35

# print information of all persons after modification
person1.print_info()
person2.print_info()
```

Output :

```
Name: Suresh, Age: 33
Name: Venkat, Age: 29
Name: Raju, Age: 35
Name: Venkat, Age: 29
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Instance variables are defined and initialized using a constructor with self parameter. Also, to access instance variables, we need instance methods with self as first parameter. It is possible that the instance methods may have other parameters in addition to the *self* parameter. To access the instance variables, we can use self.variable as shown in program. It is also possible to access the instance variables from outside the class, as: instancename.variable, e.g. person1.name

(b) **Class Variables or Static Variables :** Class variables are the variables whose single copy is available to all the instances of the class. Class variables (also known as static variables) are variables that are shared among all instances of a class. They are defined within the class, but outside of any instance methods. Class variables are typically used to store data that is shared by all instances of a class, such as constants or default values.

If we modify the copy of class variable in an instance, it will modify all the copies in the other instances. For example, if x is a class variable and if we create 3 instances, the same copy of x is passed to these 3 instances. When we modify the copy of x in any instance using a class method, the modified copy is sent to the other two instances.

EXAMPLE :

A Python program to understand class variables or static variables.

```
class Car:
    wheels = 4  # class variable
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

# create cars
car1 = Car("Toyota", "Corolla", 2010)
car2 = Car("Honda", "Civic", 2015)

# print number of wheels for all cars
print(f"{car1.make} {car1.model} has {Car.wheels} wheels")
print(f"{car2.make} {car2.model} has {Car.wheels} wheels")

# modify number of wheels for all cars
Car.wheels = 3
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL


```
# print number of wheels for all cars after modification
print(f"{car1.make} {car1.model} has {Car.wheels} wheels")
print(f"{car2.make} {car2.model} has {Car.wheels} wheels")
```

Output :

```
Toyota Corolla has 4 wheels
Honda Civic has 4 wheels
Toyota Corolla has 3 wheels
Honda Civic has 3 wheels
```

3.2 DIFFERENT TYPES OF INHERITANCE

Inheritance is a feature of Object Oriented Programming. It is used to specify that one class will get most or all of its features from its parent class. It is a very powerful feature which facilitates users to create a new class with a few or more modification to an existing class. The new class is called child class or derived class and the main class from which it inherits the properties is called base class or parent class.

The child class or derived class inherits the features from the parent class, adding new features to it. It facilitates re-usability of code.

Python Inheritance Terminologies

1. **Superclass :** The class from which attributes and methods will be inherited.
2. **Subclass :** The class which inherits the members from superclass.
3. **Method Overloading :** Redefining the definitions of methods in subclass which was already defined in superclass.

Software development is a team effort. Several programmers will work as a team to develop software.

- When a programmer develops a class, he will use its features by creating an instance to it. When another programmer wants to create another class which is similar to the class already created, then he need not create the class from the scratch. He can simply use the features of the existing class in creating his own class.
- Deriving new class from the super class is called inheritance.
- The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class.
- A child class can also override data members and methods from the parent.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

- In this process of inheritance, the base class remains unchanged. The concept of inheritance is therefore, frequently used to implement the **"is-a"** relationship.

Syntax :

```
class BaseClass:
        Body of base class
class Derivedclass(BaseClass):
Body of derived class
```

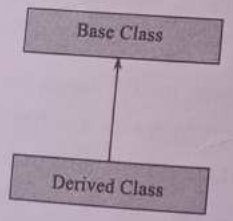
Advantages of Inheritance : The following are the few advantages or benefits of using inheritance in Python.

- Python Inheritance provides Code reusability, readability, and scalability.
- Python Inheritance reduces the code repetition. You can place all the standard methods and attributes in the parent class. These are accessible by the child class derived from it.
- By dividing the code into multiple classes, the applications look better, and the error identification is easy.

Types of Inheritance : Python supports five types of inheritance namely,

1. Single inheritance
2. Multiple inheritance
3. Multi-level inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

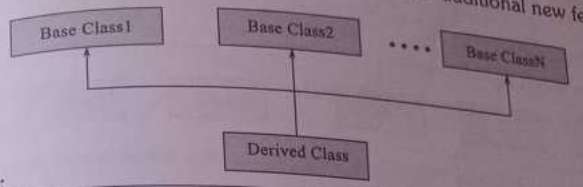
1.Single Inheritance : In single inheritance, a class can be derived from a single base class.



Syntax :

```
class  BaseClass:
           Statement block
class  Derivedclass(BaseClass):
           Statement block
```

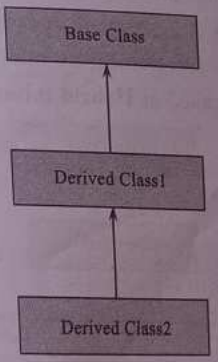

2. Multiple Inheritance : When a derived class inherits features from more than one base class it is called **mutiple inheritance.** The derived class has all the features of both the base classes and in addition to then, can have additional new features.



Syntax :

```
class  BaseClass1:
           Statement block
class  BaseClass2:
           Statement block
class  Derivedclass1(BaseClass1, BaseClass2):
           Statement block
```

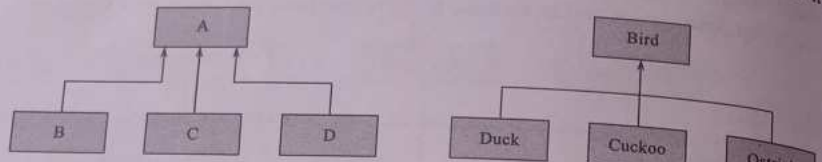
3. Multi-level Inheritance : The technique of deriving a class from an already derived class is called **multi-level inheritance.**



Syntax :

```
Class    Baseclass:
             Statement block
Class    DerivedClass1 (BaseClass) :
             Statement block
Class    Derived Class2 (DerivedClass1) :
             Statement Block
```

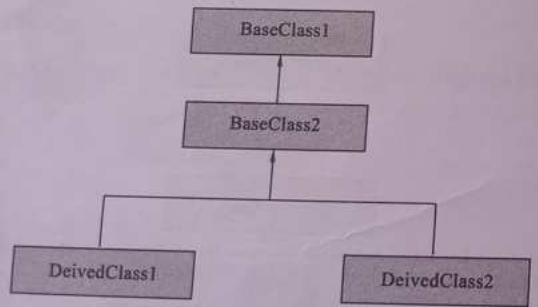
4. **Hierarchical Inheritance :** In this type of inheritance, two or more child classes inherit all the properties of the same parent class.



Syntax :

```
class BaseClass:
        Statement block
class Derivedclass1(BaseClass):
        Statement block
class Derivedclass2(BaseClass):
        Statement block
class Derivedclass3(BaseClass):
        Statement block
```

5. **Hybrid Inheritance :** Python provides us the hybrid inheritance for handling more than one type of inheritance in a program. A combination of more than one type of inheritance i.e. multiple and hierarchical or multilevel and hierarchical inheritances work together in a program called as **Hybrid inheritance.**



Syntax :

```
class BaseClass1:
        Statement block
class BaseClass2(BaseClass1):
        Statement block
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL


```
class Derivedclass1(BaseClass2):
        Statement block
class Derivedclass2(BaseClass2):
        Statement block
```

EXAMPLE-1

Python program that demonstrates Single Inheritance

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

    def display(self):
        print(f"I am {self.name} and I am {self.age} years old")

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

    def display(self):
        super().display()
        print(f"My salary is {self.salary}")

e = Employee("Suresh", 33, 55000)
e.display()
```

Output:

```
I am Suresh and I am 33 years old
My salary is 55000
```

EXAMPLE-2

Python program that demonstrates Multiple Inheritance

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

    def display(self):
        print(f"I am {self.name} and I am {self.age} years old")
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
class Student:
    def __init__(self, roll_no, marks):
        self.roll_no = roll_no
        self.marks = marks

    def display(self):
        print(f"My roll number is {self.roll_no} and my marks are {self.marks}")
class Employee(Person, Student):
    def __init__(self, name, age, roll_no, marks, salary):
        Person.__init__(self, name, age)
        Student.__init__(self, roll_no, marks)
        self.salary = salary

    def display(self):
        super().display()
        Student.display(self)
        print(f"My salary is {self.salary}")
e = Employee("Venkat", 29, 101, 90, 55000)
e.display()
```

Output:

```
I am Venkat and I am 29 years old
My roll number is 101 and my marks are 90
My salary is 55000
```

EXAMPLE-3

Python program that demonstrates Multilevel Inheritance

```
# Define base class as 'Student'
class Student:
    def getStudent(self):
        self.name = input("Enter Name: ")
        self.age = input("Enter Age: ")
        self.gender = input("Enter Gender: ")
# Define a class as 'Test' and inherit base class 'Student'
class Test(Student):
    def getMarks(self):
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL


```
        self.stuYearSem = input("Enter your Year and Semester: ")
        print("Enter the marks of the respective subjects")
        self.sub1 = int(input("IME: "))
        self.sub2 = int(input("Java: "))
        self.sub3 = int(input("Python: "))
        self.sub4 = int(input("Android Programming: "))
        self.sub5 = int(input("CNS: "))
# Define a class as 'Marks' and inherit derived class 'Test'
class Marks(Test):
    # Method
    def display(self):
        print("\n\nName: ",self.name)
        print("Age: ",self.age)
        print("Gender: ",self.gender)
        print("Study in: ",self.stuYearSem)
        print("Total Marks: ", self.sub1 + self.sub2 + self.sub3 + self.sub4 + self.sub5)
        print("Average Marks: ", (self.sub1 + self.sub2 + self.sub3 + self.sub4 + self.sub5)/5)
m1 = Marks()
# Call base class method 'getStudent()'
m1.getStudent()
# Call first derived class method 'getMarks()'
m1.getMarks()
# Call second derived class method 'display()'
m1.display()
```

Output :

```
Enter Name: Vishnu
Enter Age: 19
Enter Gender: Male
Enter your Year and Semester: III Year V-Semester
Enter the marks of the respective subjects
IME: 92
Java: 78
Python: 82
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
Android Programming: 67
CNS: 89

Name: Vishnu
Age: 19
Gender: Male
Study in: III Year V-Semester
Total Marks: 408
Average Marks: 81.6
```

EXAMPLE-4

Python program that demonstrates Hierarchical Inheritance

```
class First:
    def __init__(self):
        self.x = 20
        self.y = 10

class Second(First):
    def findsum(self):
        self.z = self.x + self.y
        print("Sum is:", self.z)

class Third(First):
    def findsub(self):
        self.z = self.x - self.y
        print("Subtraction is:", self.z)

obj1 = Second()
obj1.findsum()

obj2 = Third()
obj2.findsub()
```

Output:

```
Sum is: 30
Subtraction is: 10
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

EXAMPLE-5

Python program that demonstrates Hybrid Inheritance

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

    def eat(self):
        print(f"{self.name} is eating")

class Mammal(Animal):
    def __init__(self, name):
        super().__init__(name)

    def walk(self):
        print(f"{self.name} is walking")

class Bird(Animal):
    def __init__(self, name):
        super().__init__(name)

    def fly(self):
        print(f"{self.name} is flying")

class Bat(Mammal, Bird):
    def __init__(self, name):
        super().__init__(name)

b = Bat("Batty")
b.eat()
b.walk()
b.fly()
```

Output :

```
Batty is eating
Batty is walking
Batty is flying
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Another Example that demonstrates Hybrid Inheritance

```
class A:
    def method_A(self):
        print("This is from class A.")

class B(A):
    def method_B(self):
        print("This is from class B.")

class C(A):
    def method_C(self):
        print("This is from class C.")

class D(B, C):
    def method_D(self):
        print("This is from class D.")

# create an object of class D
d = D()

# call methods from class A, B, C, and D
d.method_A()
d.method_B()
d.method_C()
d.method_D()
```

Output:

```
This is from class A.
This is from class B.
This is from class C.
This is from class D.
```

3.3 SUPER() TO CALL METHODS AND CONSTRUCTORS OF A SUPER CLASS

To understand about python super function, you have to know about Inheritance. In Inheritance, the subclasses inherit from the superclass.

super() function/method allows us to refer the superclass implicitly. So, super() makes our task easier and comfortable. While referring the superclass from the subclass, we dont need to write the name of superclass explicitly.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

- super() is a built-in method which is useful to call the super class constructor or methods from the sub class.
- Any constructor written in the super class is not available to the sub class if the sub class has a constructor.
- So we initialize the super class instance variables and use them in the sub class by calling super class constructor using super() method inside the sub class constructor.
- super() is a built-in method which contains the history of super class methods.
- Hence, we can use super() to refer to super class constructor and methods from a sub class. So, super( ) can be used as :

```
super().init() # call super class constructor
super().init(arguments) # call super class constructor and pass arguments
super().method() # call super class method
```

EXAMPLE-1

super() method to call base class constructor example

Here's an example of using super() to call the Person superclass constructor from the Student subclass:

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

class Student(Person):
    def __init__(self, name, age, grade):
        super().__init__(name, age)  # call superclass constructor
        self.grade = grade

student = Student("Shiva", 17, A)
print(student.name)
print(student.age)
print(student.grade)
```

In this above example, we have a Person superclass and a Student subclass. The Person class has a constructor that takes a name parameter and an age parameter, and initializes the corresponding instance variables. The Student class overrides the constructor to take a name parameter, an age parameter, and a grade parameter, and it calls the

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

constructor of its superclass using super (super().__init__(name, age)) to initialize the name and age instance variables.

The output of the above code is

```
Shiva
17
A
```

EXAMPLE-2

super() method to call base class method example

```
class Animal:
    def make_sound(self):
        print("Generic animal sound")

class Dog(Animal):
    def make_sound(self):
        super().make_sound()  # call superclass method
        print("Bark!")

dog = Dog()
dog.make_sound()
```

Output :

```
Generic animal sound
Bark!
```

EXAMPLE-3

super function with multilevel inheritance

As we have stated previously that super() function allows us to refer the superclass implicitly. But in the case of multi-level inheritances super() will always refer the immediate superclass.

Also super() function not only can refer the __init__() function but also can call all other function of the superclass. So, in the following example, we will see that.

```
class A:
    def __init__(self):
        print('Initializing: class A')
    def sub_method(self, b):
        print('Printing from class A:', b)
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL


```
class B(A):
    def __init__(self):
        print('Initializing: class B')
        super().__init__()
    def sub_method(self, b):
        print('Printing from class B:', b)
        super().sub_method(b + 1)

class C(B):
    def __init__(self):
        print('Initializing: class C')
        super().__init__()
    def sub_method(self, b):
        print('Printing from class C:', b)
        super().sub_method(b + 1)

c = C()
c.sub_method(1)
```

Output:

```
Initializing: class C
Initializing: class B
Initializing: class A
Printing from class C: 1
Printing from class B: 2
Printing from class A: 3
```

So, from the output we can clearly see that the __init__() function of class C had been called at first, then class B and after that class A. Similar thing happened by calling sub_method() function.

EXAMPLE-4

Write a python program to access base class constructor and method in the sub class using super() function

```
class Square:
    def __init__(self, x = 0):
        self.x = x
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
    def area(self):
        print("Area of square:", self.x * self.x)
class Rectangle(Square):
    def __init__(self, x = 0, y = 0):
        super().__init__(x)
        self.y = y
    def area(self):
        super().area()
        print("Area of Rectangle:", self.x * self.y)
r = Rectangle(5,16)
r.area()
```

Output :

```
Area of square: 25
Area of Rectangle: 80
```

3.4 PYTHON IDENTITY OPERATORS

Python Identity operators are used to compare the memory locations of two objects, i.e., they compare whether two objects refer to the same memory location or not. There are two Identity Operators in Python: "is" and "is not".

The syntax for using Identity Operators in Python is as follows:

```
object1 is object2
object1 is not object2
```

Here's an example to illustrate the use of Python Identity Operators:

```
a = [1, 2, 3]
b = [1, 2, 3]
c = a
print("a's memory location is:" , id(a))
print("b's memory location is:" , id(b))
print("c's memory location is:" , id(c))
print(a is b)      # False - a and b refer to different memory locations
print(a is not b)  # True - a and b are not the same object
print(a is c)      # True - a and c refer to the same memory location
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Output :

```
a's memory location is: 2477652767104
b's memory location is: 2477605696640
c's memory location is: 2477652767104
False
True
True
```

In the above example, we have created two lists a and b containing the same elements. Since a and b are two different objects with their own memory locations, a is b returns False, and a is not b returns True. On the other hand, we assigned the list a to a new variable c, and c refers to the same memory location as a. Therefore, a is c returns True.

Another example involving identity operators is as follows:

```
x = 10
y = 10
print("x's memory location is:" , id(x))
print("y's memory location is:" , id(y))
print(x is y)      # True - x and y refer to the same memory location
print(x is not y)  # False - x and y are the same object
```

Output :

```
x's memory location is: 140718721987656
y's memory location is: 140718721987656
True
False
```

In this example, we have two integer variables x and y with the same value. Since integers are immutable objects in python, small integers (like 10 in this case) are stored in a shared memory location to save memory. Therefore, x is y returns True, and x is not y returns False.

3.5 CREATE AND IMPORT MODULES AND PACKAGES

Modular programming is a programming design technique that involves breaking a program down into smaller, independent modules or subtasks, which can be developed, tested, and maintained separately. Individual modules can then be clubbed together to create a larger application.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

The from...import * Statement :

It is also possible to import all the names from a module into the current namespace by using the following import statement

```
from module_name import *
```

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

There is even a variant to import all names that a module defines:

```
>>> from my_module import *
>>> print(add(34,56))
90
>>> greet("Suresh")
Hello, Suresh!
```

Import module as object

Python modules can be imported as objects. In such case, instead of module_name.function_name( ) we use object.function_name( ).

Here is the example.

```
>>> import my_module as m
>>> m.greet("Suresh")
Hello, Suresh!
>>> m.add(10, 20)
30
```

Here's another example of a python module that defines functions to display Fibonacci and factorial numbers:

 # fibonacci_fact.py

```
def fibonacci(n):
    """
    Generate the first n Fibonacci numbers and return them as a list.
    """
    fib_list = [0, 1]
    for i in range(2, n):
        fib_list.append(fib_list[i-1] + fib_list[i-2])
    return fib_list
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL


```
def factorial(n):
    """
    Calculate the factorial of n and return it.
    """
    fact = 1
    for i in range(1, n+1):
        fact *= i
    return fact
```

In this module, we define two functions: fibonacci() and factorial(). The fibonacci() function generates the first n Fibonacci numbers and returns them as a list, while the factorial() function calculates the factorial of n and returns it.

To use this module, you can create a new Python program and import the fibonacci module:

 # main.py

```
import fibonacci_fact as f

# Generate the first 10 Fibonacci numbers
fib_list = f.fibonacci(10)
print(f"Fibonacci numbers: {fib_list}")

# Calculate the factorial of 5
fact = f.factorial(5)
print(f"Factorial of 5: {fact}")
```

In this program, we import the fibonacci_fact module and use its fibonacci() and factorial() functions to generate the first 10 Fibonacci numbers and calculate the factorial of 5.

When you run this program, you should see the following output:

```
Fibonacci numbers: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Factorial of 5: 120
```

Note that the fibonacci module and the main program should be in the same directory or in a directory in Python's search path so that the module can be imported.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Here are some advantages of modular programming :

Code Reusability : Modular programming allows code to be reused in different parts of the program or even in different programs altogether. This not only saves development time and effort but also helps to improve the quality and consistency of the code.

Easy Maintenance : Modular programs are easier to maintain because each module is independent and can be updated or replaced without affecting other parts of the program. This reduces the chances of introducing errors and makes it easier to debug the code.

Improved Readability : By breaking a program into smaller modules, it becomes easier to understand and read the code. Each module can be viewed as a separate function or unit, which helps to improve the overall structure and organization of the code.

Scalability : Modular programming allows a program to be easily scaled up or down by adding or removing modules. This makes it easier to adapt the program to changing requirements or to extend its functionality in the future.

Testing and Debugging: Modular programming makes it easier to test and debug a program because each module can be tested independently. This helps to isolate errors and makes it easier to identify and fix them.

Functions, modules and packages are all constructs in python that uses code modularization.

1. **Modules :** A **module** allows you to logically organize your python code. Grouping related code into a module makes the code easier to understand and use.
   - Simply, a module is a file consisting of Python code. A module can define functions, classes and variables.
   - Modules can import other modules. It is better to place all import statements at the beginning of a module.

Here's an example of how to create and use a Python module:

1. Create a file named **my_module.py** with the following code:

```
def greet(name):
    print(f"Hello, {name}!")
def add(a, b):
    return a + b
my_variable = "This is a variable in my_module"
```

This module defines two functions: greet() and add(), as well as a variable my_variable.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

2. Save the my_module.py file in a directory of your choice.
3. The import Statement :

You can use any Python source file as a module by executing an import statement in some other python source file. The import has the following syntax

```
import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module my_module.py, create another Python file named **import_module.py** in the same directory as the my_module.py file, with the following code:

```
import my_module
my_module.greet("Suresh")
print(my_module.add(12, 23))
print(my_module.my_variable)
```

In this code, we import the my_module module and call its greet() and add() functions. We also print the my_variable variable defined in the module.

4. Run the my_program.py file. You should see the following output :

```
Hello, Suresh!
35
This is a variable in my_module
```

The from...import Statement

Python's from statement lets you import specific attributes from a module into the current namespace. The from...import has the following syntax

```
from module_name import name1[, name2[, ... nameN]]
```

For example, to import the functions greet() and add() and my_variable from the module my_module, use the following import statement that imports names from a module directly into the importing module's symbol table. For example:

```
>>> from my_module import greet, add, my_variable
>>> greet("Suresh")
Hello, Suresh!
>>> print(my_variable)
This is a variable in my_module
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

PACKAGES

A **package** is a way of organizing related modules together. A package is simply a directory that contains one or more Python modules, along with a special file called **__init__.py**. The __init__.py file is executed when the package is imported, and it can contain initialization code, as well as import statements for the modules in the package.

2 Packages are used to organize and manage large python projects. By grouping related modules together in packages, it is easier to manage and maintain the code.

Here is an example of a package called my_package:

```
my_package/
    __init__.py
    module1.py
    module2.py
```

In this example, my_package is a directory that contains two python modules, module1.py and module2.py, as well as an __init__.py file.

The __init__.py file can contain initialization code, as well as import statements for the modules in the package. For example, the __init__.py file could contain the following code:

```
from .module1 import *
from .module2 import *
```

This code imports all the names defined in module1.py and module2.py into the my_package namespace.

To use this package in your Python code, you would use the following import statement:

```
import my_package
```

is code tells Python to import the my_package package. Once the package is imported, you can access the modules in the package using dot notation:

```
import my_package.module1
import my_package.module2
```

This code imports the module1.py and module2.py modules from the my_package package.

You can also use the **from ... import ... statement** to import specific names from the modules in the package:

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL


```
from my_package.module1 import foo
from my_package.module2 import bar
```

This code imports the foo function from module1.py and the bar function from module2.py into the current namespace.

We organize a large number of files in different folders and subfolders based on some criteria, so that we can find and manage them easily. In the same way, a package in python takes the concept of the modular approach to next logical level. As you know, a module can contain multiple objects, such as classes, functions, variables etc. A package can contain one or more relevant modules. Physically, a package is actually a folder containing one or more module files.

Let's create a package named mypackage, using the following steps:

Create a new folder named D:\MyApp. Inside MyApp, create a subfolder with the name 'mypackage'. Create an empty __init__.py file in the mypackage folder.

Using a Python-aware editor like IDLE, create modules greet.py and functions.py with following code:

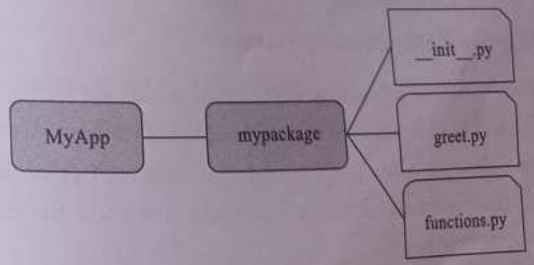
greet.py

```
def SayHello(name):
    print("Hello " + name)
    return
```

functions.py

```
def sum(x,y):
    return x+y
def average(x,y):
    return (x+y)/2
def power(x,y):
    return x**y
```

That's it. We have created our package called mypackage. The following is a folder structure:



WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Importing a Module from a Package

Now, to test our package, invoke the Python prompt from the MyApp folder.

```
D:\MyApp>python
```

Import the functions module from the mypackage package and call its power() function.

```
>>> from mypackage import functions
>>> functions.power(3,2)
9
```

It is also possible to import specific functions from a module in the package

```
>>> from mypackage.functions import sum
>>> sum(10,20)
30
>>> average(10,12)
Traceback (most recent call last):
File "<pyshell#13>", line 1, in <module>
NameError: name 'average' is not defined
```

Creating and importing Packages using classes

To tell python that a particular directory is a package, we create a file named __init__.py inside it and then it is considered as a package and we may create other modules and sub-packages within it. This __init__.py file can be left blank or can be coded with the initialization code for the package.

To create a package in Python, we need to follow these three simple steps:

1. First, we create a directory and give it a package name, preferably related to its operation.
2. Then we put the classes and the required functions in it.
3. Finally we create an __init__.py file inside the directory, to let Python know that the directory is a package.

Example of Creating Package using classes

Let's say we want to create a package that contains classes for performing basic math operations. We can call the package **mymath** and create two classes, **Calculator** and **Geometry**. The Calculator class will contain methods for performing basic arithmetic operations, and the Geometry class will contain methods for calculating area and perimeter of geometric shapes.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Here are the steps to create this package :

1. Create a directory called **mymath** in your project directory.
2. Create two Python module files called **calculator.py** and **geometry.py** in the mymath directory. These files will contain the classes for the package.
3. Open the calculator.py file and define the **Calculator** class.

```
class Calculator:
    def add(self, a, b):
        return a+b
    def subtract(self, a, b):
        return a-b
    def multiply(self, a, b):
        return a*b
    def divide(self, a, b):
        return a/b
```

4. Open the geometry.py file and define the **Geometry** class.

```
class Geometry:
    def rectangle_area(self, width, height):
        return width * height
    def rectangle_perimeter(self, width, height):
        return 2 * (width + height)
    def circle_area(self, radius):
        return 3.14159 * radius ** 2
    def circle_circumference(self, radius):
        return 2 * 3.14159 * radius
```

5. Create an empty **__init__.py** file in the mymath directory. This file is required to make the directory a Python package.
6. Test the package by importing the classes and creating instances of them in a new Python file. For example, create a file called main.py in the project directory with the following code:

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
from mymath.calculator import Calculator
from mymath.geometry import Geometry

calc = Calculator()
geo = Geometry()

print(calc.add(2, 3)) # Output: 5
print(geo.rectangle_area(5, 10)) # Output: 50
```

7. Run the main.py file to see if the package is working as expected.

THIRD PARTY PACKAGES :

The Python has got the greatest community for creating python packages. There are more than 200,000 Python packages in the world (and that's just counting those hosted on PyPI, the official Python Package Index) available at https://pypi.python.org/pypi

Python Package is a collection of all modules connected properly into one form and distributed PyPI, the Python Package Index maintains the list of Python packages available. Now when you are done with pip setup Go to command prompt / terminal and say

```
pip install <package_name>
```

Some popular Python packages that are widely used in the industry include :

- **NumPy :** A library for working with arrays and matrices of numerical data.
- **SciPy :** It is a library of algorithms and mathematical tools for python and has caused many scientists to switch from ruby to python.
- **Pandas :** It is a popular third-party Python package used for data manipulation and analysis. It provides powerful tools for working with tabular data, such as data frames, and series.
- **Matplotlib :** A plotting library that provides a wide range of visualization options for data. It can create line charts, bar charts, scatter plots, histograms, and more.
- **Requests :** It is popular third-party package in Python for making HTTP requests.
- **Scrapy :** It is a popular Python package used for web scraping and crawling websites.
- **Pillow :** It is a popular third-party Python Imaging Library (PIL) package that provides support for working with images.
- **SQLAchemy :** It is a popular Python package used for working with databases.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

- **BeautifulSoup :** It provides a powerful and flexible tool for parsing HTML and XML documents and extracting useful information.
- **Scikit-learn :** A machine learning library that provides a wide range of algorithms for supervised and unsupervised learning, as well as tools for data preprocessing and model evaluation.
- **TensorFlow :** A machine learning library developed by Google that is widely used for building and training neural networks.
- **Django:** A popular web framework for building web applications.
- **Flask :** A lightweight web framework that is ideal for building small to medium-sized web applications.

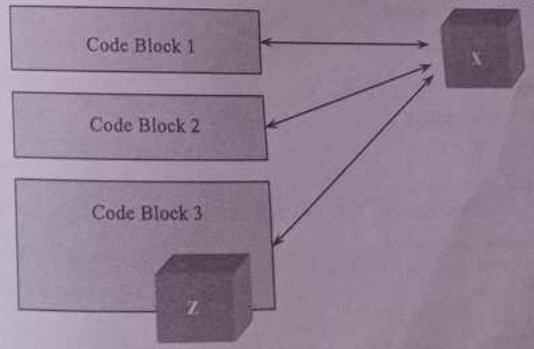
Differences between Python Modules and Packages

So, now that weve seen both modules and packages, lets see how they differ:

1. A module is a file containing python code. A package, however, is like a directory that holds sub-packages and modules.
2. A package must hold the file __init__.py. This does not apply to modules.
3. Modules are imported using the import statement followed by the name of the module. Packages, on the other hand, are imported using the import statement followed by the name of the package, and then the name of the module within the package.
4. To import everything from a module, we use the wildcard *. But this does not work with packages.

3.6 LOCAL AND GLOBAL VARIABLES

There are two types of variables: **global variables** and **local variables**. A global variable can be reached anywhere in the code, a local only in the scope.



WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

In above figure a **global variable** (x) can be reached and modified anywhere in the code; **local variable** (z) exists only in code block 3.

Local variables : Local variables are those that are defined within a function or code block and can only be accessed within that function or block. These variables are created when the function is called and are destroyed when the function returns. Local variables can have the same name as global variables, but the local variable will only be accessible within the function or block where it is defined.

Example :

```
def my_function():
    x = 10
    print("The value of x is:", x)
my_function()
print(x)
```

Output :

```
The value of x is: 10
Traceback (most recent call last):
  File "sample.py", line 6, in <module>
    print(x)
NameError : name 'x' is not defined
```

The error showing because the variable x has only local scope.

Moreover, if a variable with same name is defined inside the scope of function as well outside then it will print the value given inside the function only and not the global value.

```
def f():
    x, msg = 10, "Local scope..."
    print(x, msg)
# Global scope
x, msg = 25, "Global scope..."
f()
print(x, msg)
```

Output :

```
10 Local scope...
25 Global scope...
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Global variables : Global variables, on the other hand, are those that are defined outside of any function or block and can be accessed and modified anywhere in the program. Global variables are created when they are first assigned a value and remain in memory for the duration of the program. Here is an example of a global variable:

```
myGlobal = 5
def func1():
    myGlobal = 50
    print(myGlobal)
def func2():
    global myGlobal
    myGlobal = 156
    print(myGlobal)
def func3():
    print(myGlobal)
func1()
func3()
print("After using global keyword")
func2()
func3()
```

In this example, the variable myGlobal is defined outside of all the functions and is a global variable. To modify the global variable within the function, the global keyword is used to indicate that the variable should be treated as a global variable. The value of myGlobal can be accessed and modified within the function, and the updated value will be available outside of the function as well.

Output:

```
50
5
After using global keyword
156
156
```

Using Local and global variables in the same program :

Local and global variables can be used together in the same program. Try to determine the output of this program:

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
z = 10  #global variable
def func1():
  global z
  z = 3
def func2(a, b):
  global z
  x,y = a,b  #local variables
  return x+y+z
func1()
total = func2(4,5)
print(total)  #Output 12
```

Output:

```
12
```

3.7 VIRTUAL ENVIRONMENT IN PYTHON APPLICATION

A **Virtual Environment** is a python environment that is an isolated working copy of python which allows you to work on a specific project without affecting other projects. So basically it is a tool that enables multiple side-by-side installations of python, one for each project.

A virtual environment is a tool that helps to keep dependencies required by different projects separate by creating isolated python virtual environments for them. This is one of the most important tools that most of the Python developers use.

It is often useful to have one or more python environments where you can experiment with different combinations of packages without affecting your main installation. Python supports this through virtual environments. The virtual environment is a copy of an existing version of python with the option to inherit existing packages.

CREATING VIRTUAL ENVIRONMENT IN LINUX

1. If pip is not in your system

   $ sudo apt-get install python-pip

2. Then install virtualenv

   $ pip install virtualenv

3. Now check your installation

   $ virtualenv —version

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

4. Create a virtual environment now,

   $ virtualenv virtualenv_name

5. After this command, a folder named virtualenv_name will be created. You can name anything to it. If you want to create a virtualenv for specific python version, type

   $ virtualenv -p /usr/bin/python3 virtualenv_name

   or

   $ virtualenv -p /usr/bin/python2.7 virtualenv_name

6. Now at last we just need to activate it, using command

   $ source virtualenv_name/bin/activate

   Now you are in a Python virtual environment

7. You can deactivate virtual environment using

   $ deactivate

CREATING PYTHON VIRTUALENV IN WINDOWS

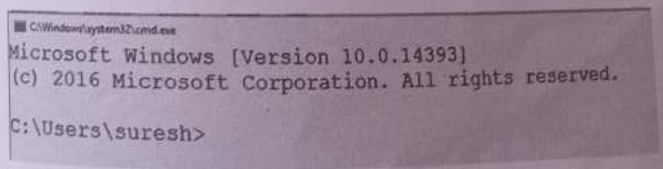
If python is installed in your system, then pip comes in handy.

So simple steps are:

1. Open a terminal
2. Check python installed on your system or not
3. Setup the pip package manager
4. Install the virtualenv package
5. Create the virtual environment
6. Activate the virtual environment
7. Install packages in virtual environment
8. Deactivate the virtual environment

1. **Open a terminal :** The method you use to open a terminal depends on your operating system.

   Windows : Open the Windows Command Prompt



Mac OS / Linux : Open the Terminal program. This is usually found under Utilities or Accessories.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

2. Check python installed on your system or not

Make sure that you have python installed on your system. You can check the version of python by running the following command in your terminal or command prompt:

`python —version`

```
C:\Users\suresh>python --version
Python 3.11.2
```

3. Setup the pip package manager

Check to see if your Python installation has pip. Enter the following in your terminal:

`pip h`

```
C:\Users\suresh>pip -h

Usage:
  pip <command> [options]

Commands:
  install                     Install packages.
  download                    Download packages.
  uninstall                   Uninstall packages.
```

If you see the help text for pip then you have pip installed, otherwise download and install pip

4. Install the virtualenv package

Once you have confirmed that you have Python installed, you can install the virtualenv package using pip. Run the following command in your terminal or command prompt:

`pip install virtualenv`

```
C:\Users\suresh>pip install virtualenv
Collecting virtualenv
  Downloading virtualenv-20.19.0-py3-none-any.whl (8.7 MB)
     ---------------------------------------- 8.7/8.7 MB 3.7 MB/s eta 0:00:00
Collecting distlib<1,>=0.3.6
  Downloading distlib-0.3.6-py2.py3-none-any.whl (468 kB)
     ---------------------------------------- 468.5/468.5 kB 4.9 MB/s eta 0:00:00
Collecting filelock<4,>=3.4.1
  Downloading filelock-3.9.0-py3-none-any.whl (9.7 kB)
Collecting platformdirs<4,>=2.4
  Downloading platformdirs-3.0.0-py3-none-any.whl (14 kB)
Installing collected packages: distlib, platformdirs, filelock, virtualenv
Successfully installed distlib-0.3.6 filelock-3.9.0 platformdirs-3.0.0 virtualenv-20.19.0
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

5. Create the virtual environment

Next, navigate to the directory where you want to create your virtual environment. You can create a new directory if you wish. Once you're in the desired directory, run the following command to create a new virtual environment:

`virtualenv mypythonenv`

After this command, a folder named mypythonenv will be created in the command prompt folder (In above C:\Users\user).

```
C:\Users\suresh>virtualenv mypythonenv
created virtual environment CPython3.11.2.final.0-64 in 15818ms
  creator CPython3Windows(dest=C:\Users\suresh\mypythonenv, clear=False, no_vcs_ignore=False, gl
obal=False)
  seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle, via=copy, app_
data_dir=C:\Users\suresh\AppData\Local\pypa\virtualenv)
    added seed packages: pip==23.0, setuptools==67.1.0, wheel==0.38.4
  activators BashActivator,BatchActivator,FishActivator,NushellActivator,PowerShellActivator,Pyt
honActivator
```

6. Activate the virtual environment

You can activate the python environment by running the following command:

Mac OS / Linux

`source mypython/bin/activate`

Windows

`mypythonenv\Scripts\activate`

You should see the name of your virtual environment in brackets on your terminal line e.g. (mypythonenv).

```
C:\Users\suresh>mypythonenv\Scripts\activate

(mypythonenv) C:\Users\suresh>
```

7. Install packages in virtual environment

Finally, you can install any packages or dependencies that your application requires using pip. For example, you can run the following command to install the numpy package:

`pip install numpy`

```
(mypythonenv) C:\Users\suresh>pip install numpy
Collecting numpy
  Downloading numpy-1.24.2-cp311-cp311-win_amd64.whl (14.8 MB)
     ---------------------------------------- 14.8/14.8 MB 3.5 MB/s
Installing collected packages: numpy
Successfully installed numpy-1.24.2
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Python Power and Logarithmic Functions

Power and Logarithm Function	Description
exp(x)	Returns E^x. This function returns the exponential value of x, which is e raised to the power of x.
expm1(x)	Returns E^x-1. This function returns the value of e raised to the power of x, minus 1.
log(x, base)	This function returns the natural logarithm of x.
log2(x)	This function returns the base-2 logarithm of x.
log10(x)	This function returns the base-10 logarithm of x.
pow(x, y)	This function returns the value of x raised to the power of y.
sqrt(x)	This function returns the square root of x.

Python Trigonometric Functions

Trigonometric Functions	Description
asin(x)	This function returns the arcsine of x, in radians.
acos(x)	This function returns the arccosine of x, in radians.
atan(x)	This function returns the arctangent of x, in radians.
atan2(y, x)	This function returns the arctangent of y/x, in radians. This is useful for computing the angle between the positive x-axis and the point (x, y).
sin(x)	This function returns the sine of x, where x is in radians.
cos(x)	This function returns the cosine of x, where x is in radians.
tan(x)	This function returns the tangent of x, where x is in radians.

Python Hyperbolic Functions

Hyperbolic Functions	Description
sinh(x)	This function returns the hyperbolic sine of x.
cosh(x)	This function returns the hyperbolic cosine of x.
tanh(x)	This function returns the hyperbolic tangent of x.
asinh(x)	This function returns the inverse hyperbolic sine of x.
acosh(x)	This function returns the inverse hyperbolic cosine of x.
atanh(x)	This function returns the inverse hyperbolic tangent of x.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Python Angular Functions

Angular Functions	Description
degrees(x)	This function converts x from radians to degrees.
radians(x)	This function converts x from degrees to radians.

Python Special Functions

Special Functions	Description
erf(x)	This function returns the error function of x.
erfc(x)	This function returns the complementary error function of x.
gamma(x)	This function returns the gamma function of x.
lgamma(x)	This function returns the natural logarithm of the absolute value of the gamma function of x.

SQRT FUNCTION

The python sqrt function is one of the python math module functions used to find the square root of a specified expression or a specific number.

The syntax of the sqrt function in python programming language is

```
math.sqrt(number);
```

Here number can be a number or a valid numerical expression. If the number argument is a positive integer, the sqrt function returns the square root of a given value. If the number argument is Negative integer, the sqrt function returns ValueError. And if it is not a number, sqrt function returns TypeError.

EXAMPLE-1

The python sqrt function finds the square root of a given number. In this sqrt example, we are going to find the square root of different data types and display the output

```
import math

number = int(input("Enter number: "))
square_root = math.sqrt(number)
print("The square root of", number, "is", square_root)
```

Output :

```
Enter number: 141
The square root of 141 is 11.87434208037917
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

The package will be installed within the virtual environment, so it will not affect other Python projects or system-wide packages.

8. **Deactivate the virtual environment**

When you're finished working on your application, to deactivate the virtual environment and use your original Python environment, simply type the command deactivate.

```
deactivate
```

```
(mypythonenv) C:\Users\suresh>deactivate
C:\Users\suresh>
```

3.8 INSTALL PACKAGES

PIP is a package management system used to install packages from repository. You can use pip to install various software packages available on http://pypi.python.org/pypi. PIP is a recursive acronym which stands for PIP installs packages.

Installing PIP : Python 2.7.9 and later (python2 series), and Python 3.4 and later (python 3 series) already comes with pip with the installation of python. You can check the version of Python by running the following command in your terminal or command prompt:

```
python --- version
```

Installing packages : Next, open your terminal or command prompt and use the pip command to install the package.

```
pip install <package_name>
```

Suppose you want to install packages called requests (which are used to make HTTP requests) and flask, you need to issue the following commands.

```
pip install requests
pip install flask
```

This will download and install the latest version of the requests package and flask package from the Python Package Index (PyPI) and install it globally on your system.

If you want to install a specific version of a package, you can use the following command:

```
pip install package_name==version_number
```

Example :

```
pip install requests==1.5.1
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Note: pip.exe is stored under C:\Python311\Scripts, so you need to go there to install packages. Alternatively, add the whole path to PATH environment variable. This way you can access pip from any directory.

Uninstalling packages : To uninstall the package use the command below.

```
pip uninstall package_name
```

Upgrade Package : To upgrade a package, use the command

```
pip install —upgrade package_name
```

Searching for a package :

```
pip search "your query"
```

Listing installed packages

```
pip list —This command will list all the installed packages.
```

Listing outdated installed packages

```
pip list —outdated
```

There are different packages in python which we can install:

- **NumPy :** A library for working with arrays and matrices of numerical data.
- **SciPy :** It is a library of algorithms and mathematical tools for python and has caused many scientists to switch from ruby to python.
- **Pandas :** It is a popular third-party Python package used for data manipulation and analysis. It provides powerful tools for working with tabular data, such as data frames, and series.
- **Matplotlib :** A plotting library that provides a wide range of visualization options for data. It can create line charts, bar charts, scatter plots, histograms, and more.
- **Requests :** It is popular third-party package in Python for making HTTP requests.
- **Scrapy :** It is a popular Python package used for web scraping and crawling websites.
- **Pillow :** It is a popular third-party Python Imaging Library (PIL) package that provides support for working with images.
- **SQLAchemy :** It is a popular Python package used for working with databases.
- **BeautifulSoup:** It provides a powerful and flexible tool for parsing HTML and XML documents and extracting useful information.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

- **Scikit-learn :** A machine learning library that provides a wide range of algorithms for supervised and unsupervised learning, as well as tools for data preprocessing and model evaluation.
- **TensorFlow:** A machine learning library developed by Google that is widely used for building and training neural networks.
- **Django:** A popular web framework for building web applications.
- **Flask:** A lightweight web framework that is ideal for building small to medium-sized web applications.

3.9 MATHEMATICAL FUNCTIONS SQRT(), COS(), SIN(), POW(), DEGREES(), AND FABS()

The Python **math package** is a built-in module that provides various functions and constants/properties, which allows us to perform mathematical functionality. Properties and functions inside the python math library object are static. So, we can access the python math properties as math.pi and functions as math.abs (number).

Python math module Properties : The following table shows you the list of properties or constants available in Python math module.

Python Math Properties	Description
math.e	It returns the Euler s Number e, approximately equal to 2.71828
math.pi	It returns the Pie Value, approximately equal to 3.14
math.inf	This Python math property returns the Positive Infinity. You can use -math.inf to return the Negative Infinity.
math.nan	It returns Not A Number as output
math.tau	This property returns the value of tau (τ), which is 6.283185307179586. Tau is a circle constant and the value is equivalent to 2π.

Python Math Library.

Python Math Functions	Description
ceil(x)	math.ceil(x) is a function from the math module that takes a single argument x and returns the smallest integer value that is greater than or equal to x. In other words, it rounds x up to the next integer value.
copysign(x, y)	math.copysign(x, y) is a function from the math module that returns the absolute value of x with the sign of y. In other words, it returns a float value with the magnitude of x and the sign of y.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

fabs(x)	math.fabs(x) is a function from the math module that returns the absolute value of x. In other words, it returns the magnitude of x without the sign.
factorial(x)	math.factorial(x) is a function from the math module that returns the factorial of x. The factorial of a non-negative integer n is the product of all positive integers from 1 to n, that is n! = 1*2*3*...*n.
floor(x)	math.floor(x) is a function from the math module that returns the largest integer less than or equal to x. In other words, it rounds down the value of x to the nearest integer.
fmod(x, y)	math.fmod(x, y) is a function from the math module that returns the remainder of the division of x by y.
frexp(x)	This function returns the mantissa and exponent of x, as pair (m, e) where m is a float value, and e is an integer value.
fsum(Iterable)	math.fsum() function is a method that returns the sum of a iterable (tuples, sets, lists, etc.) of floating-point numbers with high precision.
gcd(x, y)	math.gcd(x, y) is a function from the math module that returns the greatest common divisor (GCD) of two numbers x and y.
isclose(x,y)	math.isclose(x, y) is a function from the math module that checks whether two floating point values are "close" to each other, within a certain tolerance range.
isfinite(x)	math.isfinite(x) is a function from the math module that checks whether a given number is finite (i.e. not infinite or NaN).
isinf(x)	math.isinf(x) is a function from the math module that checks whether a given number x is positive or negative infinity.
isnan(x)	math.isnan(x) is a function from the math module that checks whether a given number x is not a number (NaN) or not.It returns TRUE if the given number is NaN otherwise FALSE.
ldexp(x, i)	math.ldexp(x, i) is a function from the math module that returns the result of multiplying a given number x by 2 raised to the power of i (x * (2**i)).
modf(x)	math.modf(x) is a function from the math module that returns the fractional and integer parts of a gin from the math module that removes the decimal values from the specified expression and returns the integer part of a given number x.
tanc(x)	math.trunc(x) is a function from the math module that removes the decimal values from the specified expression and returns the integer part of a given number x.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

EXAMPLE-2

Given a number, check if it is a prime number or not using sqrt function.

```
import math
def is_prime(number):
    if number < 2:   # check if number is less than 2
        return False
    elif number == 2:  # check if number is equal to 2
        return True
    else:
        # use the sqrt() function to optimize the loop
        for i in range(2, int(math.sqrt(number))+1):
            if number % i == 0:   # check if number is divisible by i
                return False
        return True

n = int(input("Enter number: "))
if is_prime(n):
    print(n," is prime number")
else:
    print(n," is not prime number")
```

Output :

```
Enter number: 79
79 is prime number
```

COS FUNCTION

The python cos function is one of the Python math module function, which calculates the Trigonometry Cosine for the specified expression.

The mathematical formula behind the Python Trigonometry Cosine function is

```
cos(x) = Length of the Adjacent Side / Length of the Hypotenuse
```

Syntax of a Python cos Function

The syntax of the cos Function in Python Programming Language is

```
math.cos(number);
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Here number can be a number or a valid numerical expression for which you want to find Cosine value. If the number argument is a positive or negative number, the cos function returns the Cosine value. If the number argument is not a number, the cos function returns TypeError.

Example :

```
import math
# define an angle in radians
angle = math.pi/4   # 45 degrees
# calculate the cosine of the angle using the cos() function
cosine = math.cos(angle)
print("The cosine of", angle, "is", cosine)
```

Output :

```
The cosine of 0.7853981633974483 is 0.7071067811865476
```

 # Python program showing graphical representation of cos() function

```
import math
import numpy as np
import matplotlib.pyplot as plt

in_array = np.linspace(-(3 * np.pi), 3 * np.pi, 50)
out_array = []

for i in range(len(in_array)):
    out_array.append(math.cos(in_array[i]))
    i += 1

print("in_array : ", in_array)
print("\nout_array : ", out_array)

plt.plot(in_array, out_array, color = 'red', marker = "o")
plt.title("math.cos()")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

POW FUNCTION

The Python pow function is one of the Python math module function used to calculate the Power of the specified expression. The syntax of the math pow Function in Python Programming Language is

```
math.pow(base, exponent);
```

base: specify the base value here.

exponent: specify the exponent value or power here.

For example, if x is base value and 2 is exponent, then the math.pow(x, 2) = x^2. If the base value or exponent value argument is not a number, pow function returns TypeError.

EXAMPLE

The Python pow function returns the Power of the given number. In this pow example, we are going to find the power of different data types and display the output.

```
import math

# using the pow() function to calculate 2 to the power of 3
result = math.pow(2, 3)

# print the result
print(result) # 8
```

Implementation Cases in pow() :

X	Y	Return Value
Non-negative	Non-negative	Integer
Non-negative	Negative	Float
Negative	Non-Negative	Integer
Negative	Negative	Float

 # Python code to discuss negative and non-negative cases

```
# positive x, positive y (x**y)
print("Positive x and positive y : ",end="")
print(pow(4, 3))

print("Negative x and positive y : ",end="")
# negative x, positive y (-x**y)
print(pow(-4, 3))
```



```
print("Positive x and negative y : ",end="")
# positive x, negative y (x**-y)
print(pow(4, -3))

print("Negative x and negative y : ",end="")
# negative x, negative y (-x**-y)
print(pow(-4, -3))
```

Output :

```
Positive x and positive y : 64
Negative x and positive y : -64
Positive x and negative y : 0.015625
Negative x and negative y : -0.015625
```

DEGREES FUNCTION

The Python degrees function is one of the Python math module functions used to convert the given angle from Radians to Degrees.

The syntax of the degrees Function in Python Programming Language is

```
math.degrees(number);
```

Here number can be a number or a valid numerical expression. If the number argument is a positive or negative number, the degrees function returns the output. If the number argument is not a number, degrees function returns TypeError.

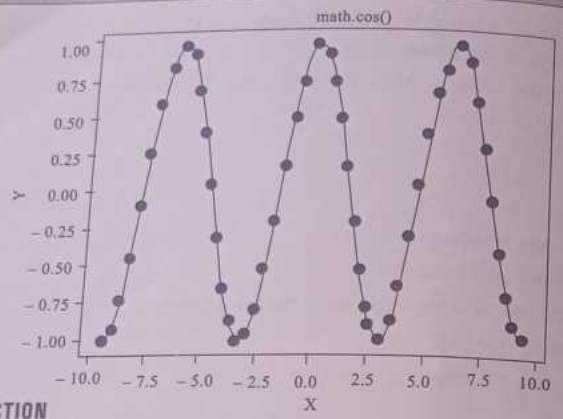
EXAMPLE

The degree Function converts the given angle from Radians to Degrees.

```
import math

# convert an angle of pi/4 radians to degrees
angle_in_radians = math.pi/4
angle_in_degrees = math.degrees(angle_in_radians)

# print the result
print(angle_in_radians, "radians is equal to", angle_in_degrees, "degrees.")
```

Output:

```
0.7853981633974483 radians is equal to 45.0 degrees.
```



SIN FUNCTION

The Python sin function is one of the Python math module function, which calculates the Trigonometry Sine value for the specified expression.

The mathematical formula behind the Python Trigonometry Sine function is

sin(x) = Length of the Opposite Side / Length of the Hypotenuse

Syntax of a sin Function

The syntax of the sin Function in Python Programming Language is

```
math.sin(number);
```

Here number can be a number or a valid numerical expression for which you want to find Sine value. If the number argument is a positive or negative number, sin function returns the Sine value. If the number argument is not a number, sin function returns TypeError.

Example

```
import math

# define an angle in radians
angle = math.pi/6   # 30 degrees
# calculate the sine of the angle using the sin() function
sine = math.sin(angle)
print("The sine of", angle, "is", sine)
```

Output:

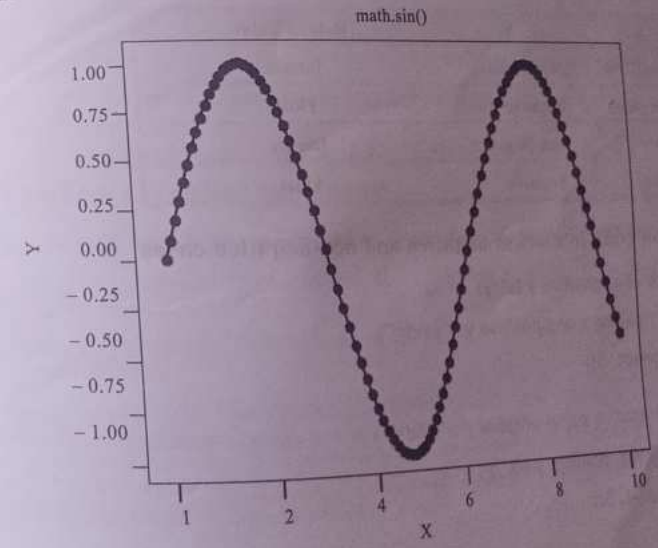
```
The sine of 0.5235987755982988 is 0.49999999999999994
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

 # Python program showing graphical representation of sin() function

```
import math
import matplotlib.pyplot as plt
import numpy as np

in_array = np.arange(0, 10, 0.1);
out_array = []

for i in range(len(in_array)):
        out_array.append(math.sin(in_array[i]))
        i += 1

print("in_array : ", in_array)
print("\nout_array : ", out_array)

# red for numpy.sin()
plt.plot(in_array, out_array, color = 'red', marker = "o")
plt.title("math.sin()")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

Output :



WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

FABS FUNCTION

The Python fabs function is one of the Python math module functions, which returns the absolute value (Positive value) of the specified expression or a specific number.

The syntax of the fabs Function in Python Programming Language to find absolute values is

```
math.fabs(number);
```

Here number can be a number or a valid numerical expression for which you want to find the absolute value in python. If the number argument is a positive or negative number, the fabs function returns the absolute value. If the number argument is not a number, the fabs function returns TypeError.

Example :

The fabs Function allows you to find the absolute values of numeric values.

```
import math

# prints the fabs using fabs() method
print ("math.fabs(-13.1) : ", math.fabs(-13.1))
print ("math.fabs(101.96) : ", math.fabs(101.96))
```

Output:

```
math.fabs(-13.1) :  13.1
math.fabs(101.96) :  101.96
```

3.10 DATETIME PACKAGE

The **datetime package** is a built-in Python module that provides classes for working with dates, times, and time intervals. It is one of the most commonly used packages in Python for dealing with time-related data and is widely used in various fields such as scientific computing, finance, and data analysis.

The datetime classes in Python are categorized into main 5 classes.

- **datetime.date:** represents a date (year, month, day).
- **datetime.time:** represents a time of day (hour, minute, second, microsecond).
- **datetime.datetime:** represents a date and time together.
- **datetime.timedelta:** represents a duration or time interval.
- **datetime.tzinfo:** represents built-in abstract base class that defines the interface for working with time zones.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

datetime is part of pythons standard library, which means, you dont need to install it separately.

You can simply import as below.

```
import datetime
```

Get Current Date and Time

```
import datetime

# get the current date and time
current_datetime = datetime.datetime.now()

# print the current date and time
print("Current date and time:", current_datetime)
```

When you run the program, the output will be something like:

```
Current date and time: 2023-02-17 12:44:40.195791
```

The output is with the current date and time in local time zone. The output is in the following order: year, month, date, hour, minute, seconds, microseconds.

In the above example, we have imported datetime module using import datetime statement. One of the classes defined in the datetime module is datetime class. We then used now() method to create a datetime object containing the current local date and time.

To get the date alone, use the datetime.date.today() in above statement.

```
datetime.date.today()
#> datetime.date(2023, 2, 17)
```

It returns a datetime.date object and not datetime.datetime.

How to create the datetime object

We saw how to create the datetime object for current time. But how to create one for any given date and time. Say, for the following time: 2001-01-31::10:51:00

You can pass it in the same order to datetime.datetime().

```
datetime.datetime(2001, 1, 31, 10, 51, 0)
#> datetime.datetime(2001, 1, 31, 10, 51)
```

You can also create a datetime from a unixtimestamp. A unixtimestamp is nothing but the number of seconds since the epoch date: Jan 01, 1970

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
mydatetime = datetime.datetime.fromtimestamp(528756281)
mydatetime
#> datetime.datetime(1986, 10, 4, 2, 14, 41)
```

You can convert the datetime back to a unixtimestamp as follows:

```
mydatetime.timestamp()
#> 528756281.0
```

Commonly used classes in the datetime module are:

1. date class
2. time class
3. datetime class
4. timedelta class

1. **datetime.date class :** The datetime.date class represents a date, without a time or time zone. It has the attributes year, month, and day, which can be set to any integer value within the range of valid dates.

EXAMPLE–1

Date object to represent a date

```
import datetime
d = datetime.date(2022, 2, 17)
print(d)
```

When you run the program, the output will be:

```
2022-02-17
```

We can only import date class from the datetime module. Here's how:

```
from datetime import date
a = date(2019, 4, 13)
print(a)  # 2019-04-13
```

EXAMPLE –2

Get current date

You can create a date object containing the current date by using a class method named today(). Here's how:

```
from datetime import date
today = date.today()
print("Current date =", today)  # Current date = 2023-02-17
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

EXAMPLE –3

Get date from a timestamp

We can also create date objects from a timestamp. A UNIX timestamp is the number of seconds between a particular date and January 1, 1970 at UTC. You can convert a timestamp to date using fromtimestamp() method.

```
from datetime import date
timestamp = date.fromtimestamp(1326244364)
print("Date =", timestamp)
```

When you run the program, the output will be:

```
Date = 2012-01-11
```

EXAMPLE –4

Print today's year, month, day and weekday

We can get year, month, day, day of the week etc. from the date object easily. Here's how:

```
import datetime

date_object = datetime.date.today()  # create a date object for todays date

# print the date object
print("Date object:", date_object)
# print the year, month, day, and weekday from the date object
print("Year:", date_object.year)
print("Month:", date_object.month)
print("Day:", date_object.day)
print("Weekday:", date_object.weekday() )  # Monday is 0 and Sunday is 6
```

Output :

```
Date object: 2023-02-17
Year: 2023
Month: 2
Day: 17
Weekday: 4
```

2. **datetime.time class :** The datetime.time class represents a time of day, without a date or a time zone. It has the attributes hour, minute, second, and microsecond, which can be set to any integer value within the range of valid times. Here's an example of how to create and use a datetime.time object:

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
import datetime
# create a time object for 9:30:45.678 using datetime.time(9, 30, 45, 678000)
time_object = datetime.datetime.now().time()  # create a time object for present time
# print the time object
print("Time object:", time_object)
# print the hour, minute, second, and microsecond attributes
print("Hour:", time_object.hour)
print("Minute:", time_object.minute)
print("Second:", time_object.second)
print("Microsecond:", time_object.microsecond)
```

Output :

```
Time object: 13:03:56.117219
Hour: 13
Minute: 3
Second: 56
Microsecond: 117219
```

3. **datetime.datetime class :** The datetime.datetime class represents a date and time, with an optional time zone. It has the attributes year, month, day, hour, minute, second, and microsecond, which can be set to any integer value within the range of valid dates and times. Here's an example of how to create and use a datetime.datetime object:

```
import datetime
# create a datetime object for February 17, 2023 at 9:30:45.678
datetime_object = datetime.datetime(2023, 2, 17, 9, 30, 45, 678000)
# print the datetime object
print("Datetime object:", datetime_object)
# print the the year, month, day, hour, minute,
# second, microsecond and timestamp attributes
print("Year:", datetime_object.year)
print("Month:", datetime_object.month)
print("Day:", datetime_object.day)
print("Hour:", datetime_object.hour)
print("Minute:", datetime_object.minute)
print("Second:", datetime_object.second)
print("Microsecond:", datetime_object.microsecond)
print("timestamp =", datetime_object.timestamp())
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

4. **datetime.timedelta class :** The datetime.timedelta class represents duration, or the difference between two dates or times. It can be used to perform arithmetic operations on dates and times, such as adding or subtracting a certain number of days, hours, minutes, or seconds. Here's an example of how to create and use a datetime.timedelta object:

```
import datetime
# create a timedelta object representing 2 days, 3 hours, and 30 minutes
delta_object = datetime.timedelta(days=2, hours=3, minutes=30)
# create a datetime object for February 17, 2023 at 9:30:45
datetime_object = datetime.datetime(2023, 2, 17, 9, 30, 45)
# add the timedelta object to the datetime object
new_datetime_object = datetime_object + delta_object

# subtract the timedelta object from the datetime object
old_datetime_object = datetime_object - delta_object
# print the results
print("Original datetime:", datetime_object)
print("New datetime:", new_datetime_object)
print("Old datetime:", old_datetime_object)
```

Output:

```
Original datetime: 2023-02-17 09:30:45
New datetime: 2023-02-19 12:00:45
Old datetime: 2023-02-14 06:00:45
```

Difference between two timedelta objects

In Python, you can subtract one datetime object from another datetime object to get a timedelta object representing the difference between the two dates or times. You can also subtract one timedelta object from another timedelta object to get a new timedelta object representing the difference between the two durations. Here's an example:

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
import datetime

# create two datetime objects
start_time = datetime.datetime(2023, 2, 17, 9, 30, 0)
end_time = datetime.datetime(2023, 2, 20, 17, 30, 0)

# calculate the difference between the two datetime objects
time_delta = end_time - start_time

# create two timedelta objects
delta1 = datetime.timedelta(hours=1)
delta2 = datetime.timedelta(days=2, hours=3)

# calculate the difference between the two timedelta objects
delta3 = delta2 - delta1

# print the results
print("Time difference:", time_delta)
print("Delta difference:", delta3)
```

Output :

```
Time difference: 3 days, 8:00:00
Delta difference: 2 days, 2:00:00
```

How to format the datetime object into any date format

The way date and time is represented may be different in different places, organizations etc. It's more common to use mm/dd/yyyy in the US, whereas dd/mm/yyyy is more common in the UK.

Python has **strftime()** and **strptime()** methods to handle this.

Python strftime() - datetime object to string

You can format a datetime object into any date format using the strftime() method, which stands for "string format time". The strftime() method takes a format string as an argument, which specifies how the date and time should be displayed. Here's an example of how to format a datetime object into a custom date format:

```
import datetime

# create a current datetime object
dt = datetime.datetime.now()

# format the datetime object as "YYYY-MM-DD"
date_string = dt.strftime("%Y-%m-%d")

# format the datetime object as "DD/MM/YYYY"
date_string2 = dt.strftime("%d/%m/%Y")

# format the datetime object as "YYYY-MM-DD HH:MM:SS"
datetime_string = dt.strftime("%Y-%m-%d %H:%M:%S")

# print the results
print(date_string)
print(date_string2)
print(datetime_string)
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

When you run the program, the output will be something like:

```
2023-02-17
17/02/2023
2023-02-17 21:40:55
```

Here, %Y, %m, %d, %H etc. are format codes. The strftime() method takes one or more format codes and returns a formatted string based on it.

```
%Y - year [0001,..., 2018, 2019,..., 9999]
%m - month [01, 02, ..., 11, 12]
%d - day [01, 02, ..., 30, 31]
%H - hour [00, 01, ..., 22, 23
%M - minute [00, 01, ..., 58, 59]
%S - second [00, 01, ..., 58, 59]
```

Python strptime() - string to datetime

The strptime() method is used to parse a string representation of a date and time into a datetime object. The strptime() method takes two arguments: the string representation of the date and time, and a format string that specifies how the date and time are formatted in the string. Here's an example of how to use the strptime() method to parse a date string into a datetime object:

```
import datetime

# create a string representing the date and time
date_string = "2023-02-17 09:30:45"

# parse the string into a datetime object
dt = datetime.datetime.strptime(date_string, "%Y-%m-%d %H:%M:%S")
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
# print the datetime object
print(dt)
```

When you run the program, the output will be:

```
2023-02-17 09:30:45
```

The format string "%Y-%m-%d %H:%M:%S" specifies that the date and time are formatted as follows:

```
%Y - 4-digit year
%m - 2-digit month (with leading zeros)
%d - 2-digit day (with leading zeros)
%H - 24-hour format hour (with leading zeros)
%M - minute (with leading zeros)
%S - second (with leading zeros)
```

EXERCISE –1

How many days has it been since you were born?

```
import datetime

# Enter your birthdate
birthdate = datetime.date(1989, 11, 27)

# Calculate the number of days since birthdate
days_since_birth = (datetime.date.today() - birthdate).days

# Print the result
print("It has been {} days since your birthdate.".format(days_since_birth))
```

Output :

```
It has been 12135 days since your birthdate.
```

EXERCISE– 2:

How to count the number of Saturdays between two dates?

```
import datetime

# Define the start and end dates
start_date = datetime.date(2022, 1, 1)
end_date = datetime.date(2022, 12, 31)

# Initialize a counter variable
num_saturdays = 0
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL


```
# Iterate over the days between the start and end dates
for d in range((end_date - start_date).days + 1):
    date = start_date + datetime.timedelta(d)

    # Check if the current day is a Saturday
    if date.weekday() == 5:
        num_saturdays += 1

# Print the result
print("There are {} Saturdays between {} and {}.".format(num_saturdays, start_date, end_date))
```

Output :

```
There are 53 Saturdays between 2022-01-01 and 2022-12-31.
```

EXERCISE–3

How many days are left until your next birthday this year?

```
import datetime

# Enter your birthdate and the current year
birthdate = datetime.date(1989, 11, 27)
current_year = datetime.date.today().year

# Calculate the date of your next birthday
next_birthday = datetime.date(current_year, birthdate.month, birthdate.day)
if next_birthday < datetime.date.today():
    next_birthday = datetime.date(current_year + 1, birthdate.month, birthdate.day)

# Calculate the number of days until your next birthday
days_left = (next_birthday - datetime.date.today()).days

# Print the result
print("There are {} days left until your next birthday.".format(days_left))
```

Output :

```
There are 283 days left until your next birthday.
```

EXERCISE –4

How to convert number of days to seconds?

```
import datetime
bdate = datetime.date(1989, 11, 27)
td = datetime.date.today() - bdate

print(td.total_seconds())
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Output :

```
1048464000.0
```

EXERCISE –5

How to format a given date to "mmm-dd, YYYY" format?

```
import datetime

date_str = "2022-02-17"

date_obj = datetime.datetime.strptime(date_str, "%Y-%m-%d")

formatted_date = date_obj.strftime("%b-%d, %Y")

print(formatted_date)
```

Output :

```
Feb-17, 2022
```

CHAPTER 4

EXCEPTION HANDLING AND MULTITHREADING

Chapter Outline

4.1 DIFFERENTIATE BETWEEN COMPILE TIME ERRORS, RUNTIME ERRORS AND LOGICAL ERRORS

Compile-time errors, runtime errors, and logical errors are all types of programming errors that can occur in software development. The errors in the software are called bugs and the processes of removing them are called debugging. In general, we can classify errors in a program into one of these three types:

(a) Compile-time errors

(b) Runtime errors

(c) Logical errors

(a) Compile-time errors : These are syntactical errors that occur during the compilation of the source code. The compiler detects these errors and stops the compilation process, preventing the executable code from being created. Examples of compile-time errors include syntax errors, missing semicolons, and type mismatches. Such errors are detected by python compiler and the line number along with error description is displayed by the python compiler.

Example :

A Python program to understand the compile-time error

```
a = 1
if a == 1
  print("hello")
```

Output :

```
File "main.py", line 2
  if a == 1
       ^
SyntaxError : invalid syntax
```

(b) Runtime errors : These are errors that occur when the program is running. Runtime errors are caused by unexpected conditions or events that arise during program execution, such as division by zero, null pointer dereferencing, or file not found errors. Runtime errors are not detected by the python compiler. They are detected by the Python Virtual Machine (PVM), only at runtime. Some of the Runtime errors are

- **NameError:** This occurs when the code tries to use a variable or function that has not been defined. For example, if you try to print a variable that has not been assigned a value.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

- **Index Error :** This occurs when the code tries to access an element of a list or string that does not exist. For example, if you try to access the fifth element of a list that only has three elements.
- **Zero Division Error :** This occurs when the code tries to divide a number by zero.
- **Type Error :** This occurs when the code tries to perform an operation on a data type that is not supported. For example, if you try to add a string to an integer.
- **Import Error :** This occurs when the python interpreter is unable to find a module that is being imported in the code.
- **Key Error :** This occurs when the code tries to access a dictionary key that does not exist.
- **Attribute Error :** This occurs when the code tries to access an attribute of an object that does not exist. For example, if you try to access an attribute that only exists in a subclass of the object.
- **File Not Found Error :** This error occurs when the program tries to read, write or manipulate a file that does not exist in the specified path or directory.

Example–1: A Python program to understand the Runtime error.

```
x = 10
y = 0
z = x / y  # raises a ZeroDivisionError: division by zero
```

Example–2:

```
x = 10
y = "hai"
z = x + y  # raises a TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Example–3:

```
my_list = [1, 2, 3]
print(my_list[3])  # raises an IndexError: list index out of range
```

(c) Logical errors : These are errors that occur when the program runs without crashing or producing any error messages, but it does not produce the expected output. Logical errors are caused by mistakes in the program's algorithm or design, and they can be difficult to identify and fix. Examples of logical errors include incorrect calculations, wrong assumptions about input data, and incorrect program flow. In the following program, the programmer wants to calculate incremented salary of an employee, but he gets wrong output, since he uses wrong formula.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Example: A Python program to increment the salary of an employee by 15%.

```
def increment(sal):
    sal = sal * 15/100
    return sal
sal = increment(5000)
print("Salary after Increment is:", sal)
```

Output::

```
Salary after Increment is: 750.0
```

From the above program the formula for salary is wrong, because only the increment but it is not adding it to the original salary. So, the correct formula would be:

```
sal = sal + sal * 15/100
```

Sometimes there can be absolutely nothing wrong with your python implementation of an algorithm the algorithm itself can be incorrect. However, more frequently these kinds of errors are caused by programmer carelessness. Here are some examples of mistakes which lead to logical errors:

- Using the wrong variable name
- Indenting a block to the wrong level
- Using integer division instead of floating-point division
- Getting operator precedence wrong
- Making a mistake in a boolean expression
- off-by-one, and other numerical errors

Compile time errors and runtime errors can be eliminated by the programmer by modifying the program source code. In case of runtime errors, when the programmer knows which type of error occurs, he has to handle them using exception handling mechanism.

COMMON COMPILE TIME AND RUNTIME ERRORS

Compile time errors

Errors that occur when you violate the rules of writing syntax are known as **Compile time errors**. This compiler error indicates something that must be fixed before the code can be compiled. All these errors are detected by the compiler and thus are known as compile-time errors.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Most frequent Compile-Time errors include :

- Syntax errors
- Missing Parenthesis ()
- Printing the value of variable without declaring it
- Missing semicolon (terminator)
- Leaving out a keyword
- Putting a keyword in the wrong place
- Leaving out a symbol, such as a colon, comma or brackets
- Misspelling a keyword
- Incorrect Indentation
- Empty block

Python will find these kinds of errors when it tries to parse your program, and exit with an error message without running anything. Many of the errors listed below can be detected by a good python editor or development interface.

- SyntaxError: invalid syntax
- SyntaxError: EOL while scanning string literal
- IndentationError

EXAMPLE-1

```
myfunction(x, y):
    return x + y
else:
    print("Hello!")

if mark >= 50
    print("You passed!")

if arriving:
    print("Hi!")
esle:
    print("Bye!")

if flag:
print("Flag is set!")
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Output :

```
File "main.py", line 1
    myfunction(x, y):
                   ^
SyntaxError: invalid syntax
```

EXAMPLE-2

```
x = int(input('Enter a number: '))
if x%2 == 0
    print('You have entered an even number.')
else:
    print ('You have entered an odd number.')
```

Output :

```
File "main.py", line 2
    if x%2 == 0
               ^
SyntaxError: expected ':'
```

Runtime errors or Exceptions

Errors can also occur at runtime and these are called **Exceptions** or **Runtime errors**. They occur, for example, when a file we try to open does not exist (FileNotFoundError), dividing a number by zero (ZeroDivisionError), module function we try to access is not found (ImportError) etc.

Whenever these type of runtime error occur, python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

```
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    1/0
ZeroDivisionError: division by zero
```

```
>>> open("imaginary.txt")
FileNotFoundError: [Errno 2] No such file or directory: 'imaginary.txt'
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

- An exception is a runtime error which can be handled by the programmer. That means if the programmer can guess an error in the program and he can do something to eliminate the harm caused by that error, then it is called an **exception handling**.
- If the programmer cannot do anything in case of an error, then it is called an **error** and not an exception.
- All exceptions are represented as classes in python. The exceptions which are already available in python are called **built-in exceptions**. The base class for all built-in exceptions is **BaseException** class.
- From BaseException class, the sub class Exception is derived. From Exception class, the sub classes StandardError and Warning are derived.
- All errors (or exceptions) are defined as sub classes of StandardError. An error should be compulsory handled otherwise the program will not execute.
- Similarly, all warnings are derived as sub classes from Warning class. A warning represents a caution and even though it is not handled, the program will execute. So, warnings can be neglected but errors cannot be neglected.
- Just like the exceptions which are already available in python language, a programmer can also create his own exceptions, called **user-defined exceptions.**
- When the programmer wants to create his own exception class, he should derive the class from Exception class and not from BaseException class.

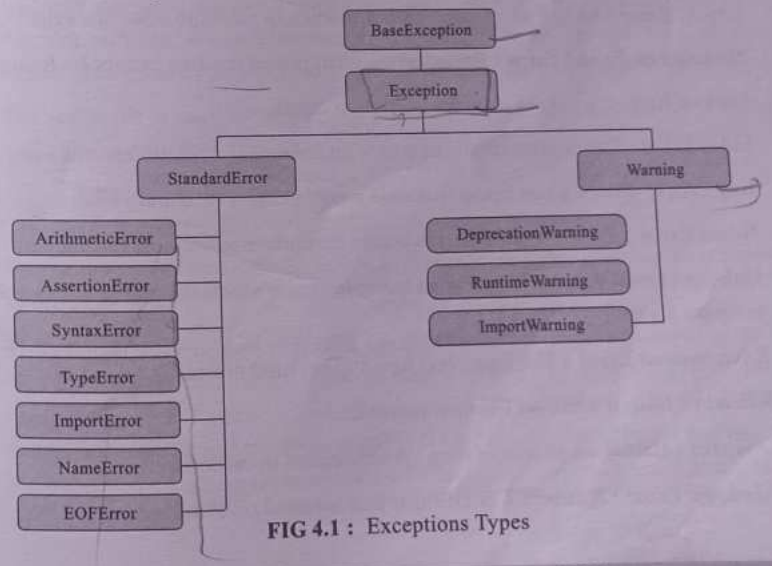


FIG 4.1 : Exceptions Types

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

List of Standard Exceptions :

Base Exception : The base class for all built-in exceptions.

System Exit : Raised when the sys.exit() function is called.

Keyboard Interrupt : Raised when the user presses Ctrl+C to interrupt the program.

Exception: The base class for all non-system-exiting exceptions.

Stop Iteration : Raised when an iterator has no more items.

Arithmetic Error : The base class for all arithmetic errors.

Floating Point Error : Raised when a floating-point calculation fails.

Overflow Error : Raised when a calculation exceeds the maximum representable value.

Zero Division Error : Raised when attempting to divide by zero.

Assertion Error : Raised when an assertion fails.

Attribute Error : Raised when an attribute reference or assignment fails.

Buffer Error : Raised when a buffer-related operation fails.

EOF Error : Raised when there is no input from the input() function and the end of the file is reached.

Import Error : Raised when an imported module or attribute does not exist.

Module Not Found Error : Raised when an imported module cannot be found.

Lookup Error : The base class for all lookup errors.

Index Error : Raised when trying to access an index that does not exist in a sequence.

Key Error : Raised when trying to access a non-existent dictionary key.

Name Error : Raised when trying to access an undefined variable or function.

Unbound Local Error : Raised when trying to access a local variable that has not been assigned a value.

Environment Error : The base class for all I/O-related errors.

IOError : Raised when an I/O operation fails.

OSError : Raised when a operating system-related operation fails.

Windows Error : A subclass of OSError that is raised on Windows.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Runtime Error : The base class for all runtime errors.

Recursion Error : Raised when the maximum recursion depth is exceeded.

Not Implemented Error : Raised when an abstract method or function is not implemented.

Syntax Error : Raised when the syntax of the code is invalid.

Indentation Error : Raised when the indentation of the code is invalid.

TabError : Raised when the indentation of the code contains tabs and spaces.

System Error : Raised when an internal error occurs in the Python interpreter.

Type Error : Raised when an operation or function is applied to an object of inappropriate type.

Value Error : Raised when a function or operation receives an argument of the correct type but an inappropriate value.

4.3 TRY, EXCEPT, FINALLY AND ELSE BLOCKS TO HANDLE EXCEPTIONS

The purpose of handling exceptions is to make the program robust. The word robust means strong. A robust program does not terminate in the middle. Also, when there is an exception in the program, it will display an appropriate message to the user and continue execution. Designing such programs is needed in any software development. For that purpose, the programmer should handle the exceptions. You can use try, except, finally, and else blocks to handle exceptions that may occur in your code.

The **try block** is used to enclose the code that may raise an exception. A try block looks like as follows:

```
try:
    statement(s)
```

If an exception occurs inside the try block, the code execution is immediately transferred to the except block.

The **except block** is used to handle the exception that occurred inside the try block. You can have multiple except blocks to handle different types of exceptions. When an exception occurs, Python looks for the first except block that can handle that type of exception and executes the code inside it. If no except block is found that can handle the exception, the program terminates with an error message. Except block looks like as follows:

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
except exceptionname:
    statement(s)
```

The statements written inside an except block are called **handlers** since they handle the situation when the exception occurs.

The **else block** is also optional and is executed if no exception occurs in the try block. This block is useful when you want to execute some code only if there is no exception. Else block looks like as follows:

```
else:
    statement(s)
```

The **finally block** is optional, and it is executed whether an exception occurs or not. This block is used to perform cleanup actions, such as closing a file or releasing a resource. Finally block looks like as follows:

```
finally:
    statements
```

Here, the complete exception handling syntax will be in the following format:

```
try:
    # code that may raise an exception
    statement(s)
except Exception1:
    # code that handles the Exception1
    statement(s)
except Exception2:
    # code that handles the Exception2
    statement(s)
else:
# execute if no exception occurs
    statement(s)
finally:
    # execute whether exception occurs or not
    satement(s)
```

The following points are followed in exception handling :

- A single try block can be followed by several except blocks.
- Multiple except blocks can be used to handle multiple exceptions.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

- We cannot write except blocks without a try block.
- We can write a try block without any except blocks.
- Else block and finally blocks are optional.
- When there is no exception, else block is executed after try block.
- Finally block is always executed.

EXAMPLE–1

The example below accepts two numbers from the user and performs their division. It demonstrates the uses of else and finally blocks.

```
try:
    print("try block")
    num1 = int(input('Enter a number: '))
    num2 = int(input('Enter another number: '))
    result = num1 / num2
    print(f"The result of {num1}/{num2} is {result}")
except ZeroDivisionError:
    print("Cannot divide by zero")
except ValueError:
    print("Please enter a valid number")
else:
    print("else block")
    print("No exception was raised")
finally:
    print("finally block")
    print("This will always execute")
print ("Out of try, except, else and finally blocks." )
```

Output :

```
First run
try block
Enter a number: 12
Enter another number: 0
Cannot divide by zero
finally block
This will always execute
Out of try, except, else and finally blocks.
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
Second run
try block
Enter a number: 10
Enter another number: 5
The result of 10/5 is 2.0
else block
No exception was raised
finally block
This will always execute
Out of try, except, else and finally blocks.

Third run
try block
Enter a number: 10
Enter another number: xyz
Please enter a valid number
finally block
This will always execute
Out of try, except, else and finally blocks.
```

EXAMPLE–2

A python program to handle IOError produced by open() function.

```
try:
    file = open("example.txt", "r")
    content = file.read()
    print(content)
    file.close()
except IOError as e:
    print("Error: File not found.", e.strerror)
```

Output :

```
Error: File not found. No such file or directory
```

In the above program if the file is not found, then IOError is raised. Then except block will display a message: Error: File not found. If the file is found, then all the lines of the file are read using read() method.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

The Except Block :

The except block is useful to catch an exception that is raised in the try block. When there is an exception in the try block, then only the except block is executed. It is written in various formats.

1. To catch the exception which is raised in the try block, we can write except block with the Exceptionclass name as:

```
except Exceptionclass:
```

2. We can catch the exception as an object that contains some description about the exception.

```
except Exceptionclass as obj:
```

3. To catch multiple exceptions, we can write multiple catch blocks. The other way is to use a single except block and write all the exceptions as a tuple inside parantheses as:

```
except (Exceptionclass1, Exceptionclass2, ...):
```

4. To catch any type of exception where we are not bothered about which type of exception it is, we can write except block without mentioning any Exceptionclass name as:

```
except:
```

EXAMPLE–3

```
try:
    f = open('myfile.txt','w')
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1/num2
    print(f"The result of {num1}/{num2} is {result}")
    s = f.write(str(result))
    print("Result is stored in file")
except ValueError:
    print("Please enter a valid number")
except ZeroDivisionError:
    print("Cannot divide by zero")
except Exception as e:
    print("An error occurred:", e)
except:
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
        print("The positive number is:", positive_num)
    except ValueError as e:
        print(e)
```

Output :

```
First run:
Enter a positive number: 25
The positive number is: 25
Second run:
Enter a positive number: -45
Enter only positive number
```

4.5 USER DEFINED EXCEPTIONS

Like built-in exceptions of python, the programmer can also create his own exceptions which are called **User-defined exceptions** or **Custom exceptions.** We know python offers many exceptions which will raise in different contexts. But, there may be some situations where none of the built-in exceptions in python are useful for the programmer. In that case, the programmer has to create his/her own exception and raise it.

For example, lets take a bank where customers have accounts. Each account is characterized by customer account number and balance amount. The rule of the bank is that every customer should keep minimum Rs. 2000.00 as balance amount in his account. The programmer now is given a task to check the account to know whether customer is maintaining minimum balance of Rs. 2000.00 or not. If the balance amount is below Rs. 2000.00, then the programmer wants to raise an exception saying Minimum balance requirement not met. Since there is no such exception available in python, the programmer has to create his/her own exception.

EXAMPLE–1

```
class MinimumBalanceError(Exception):
    """Custom exception raised when the account balance falls below the minimum balance."""
    pass

class BankAccount:
    def __init__(self, account_number, balance, minimum_balance):
        self.account_number = account_number
        self.balance = balance
        self.minimum_balance = minimum_balance
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL


```
    def withdraw(self, amount):
        if self.balance - amount < self.minimum_balance:
            raise MinimumBalanceError("Minimum balance requirement not met.")
        else:
            self.balance -= amount
            print(f"Withdrawal of {amount} successful. Available balance: {self.balance}.")

# Testing the code with different withdrawal amounts
account = BankAccount("1234567890", 5000, 2000)
try:
    account.withdraw(4000)
except MinimumBalanceError as e:
    print(e)

try:
    account.withdraw(1000)
except MinimumBalanceError as e:
    print(e)

try:
    account.withdraw(3000)
except MinimumBalanceError as e:
    print(e)
```

Output :

```
Minimum balance requirement not met.
Withdrawal of 1000 successful. Available balance: 4000.
Minimum balance requirement not met.
```

EXAMPLE–2

```
class NegativeNumberError(Exception):
    """Custom exception raised when a negative number is encountered."""
    pass

def square_root(num):
    if num < 0:
        raise NegativeNumberError("Cannot compute square root of negative number")
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
        print("Unexpected error:")
    finally:
        f.close()
```

Output :

```
Enter a number: 12
Enter another number: 3
The result of 12/3 is 4.0
Result is stored in file
```

EXAMPLE-4

```
try:
    if (3+4-5)>0:
        a=3
        a.append("hello") # throws Attribute Error
        print("hello"+4) # throws TypeError
except (AttributeError, TypeError) as e:
    print("Error occurred:", e)
finally:
    print("try except block successfully executed")
```

Output :

```
Error occurred: 'int' object has no attribute 'append'
try except block successfully executed
```

4.4 RAISE STATEMENT

The **raise** statement in python is used to raise an exception. When an exception is raised, it can be caught by an exception handler and dealt with accordingly. The basic syntax of the raise statement is as follows:

```
raise exception_class("Error message")
```

The exception_class is the type of exception that you want to raise (for example, NameError). This can be any class that inherits from the built-in Exception class or one of its subclasses. The "Error message" is a string that provides additional information about the exception. Here's an example:

```
def divide(x, y):
    if y == 0:
        raise ZeroDivisionError("Cannot divide by zero")
    return x/y
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL


```
try:
    result = divide(10, 0)
except ZeroDivisionError as error:
    print(error)
```

In this example, the divide function raises a ZeroDivisionError exception if the second argument y is zero. In the try block, the divide function is called with arguments 10 and 0, which causes the exception to be raised. The exception is caught by the except block, which prints the error message "Cannot divide by zero".

EXAMPLE-1

Input an age that raises an exception if age is less than 18

```
def check_age(age):
    if age < 18:
        raise ValueError("Age should be at least 18 years old to vote.")
    else:
        print("You are eligible for voting")

# Testing the function with different ages
age = int(input("Enter the age: "))
try:
    check_age(age)
except ValueError as e:
    print(e)
```

Output :

```
Enter the age: 17
Age should be at least 18 years old to vote.
```

EXAMPLE -2

Input a positive number and raise an exception if input is a negative value.

```
def get_positive_number():
    num = int(input("Enter a positive number: "))
    if num < 0:
        raise ValueError("Enter only positive number")
    return num

try:
    positive_num = get_positive_number()
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
    else:
        return num**0.5

# Testing the function with different values
num = int(input("Enter a positive number: "))
try:
    print(square_root(num))
except NegativeNumberError as e:
    print(e)
```

Output :

```
First run:
Enter a positive number: 17
4.123105625617661

Second run:
Enter a positive number: -17
Cannot compute square root of negative number
```

4.6 MULTITHREADING

A **thread** is the smallest unit of execution with the independent set of instructions. It is a part of the process and operates in the same context sharing programs runnable resources like memory and processor. A thread has a starting point, an execution sequence, and a result. It has an instruction pointer that holds the current state of the thread and controls what executes next in what order.

A thread is basically an independent flow of execution. A single process can consist of multiple threads. Each thread in a program performs a particular task. For Example, when you are playing a game say FIFA on your PC, the game as a whole is a single process, but it consists of several threads responsible for playing the music, taking input from the user, running the opponent synchronously, etc. All these are separate threads responsible for carrying out these different tasks in the same program.

Every process has one thread that is always running. This is the main thread. This main thread actually creates the child thread objects. The child thread is also initiated by the main thread.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

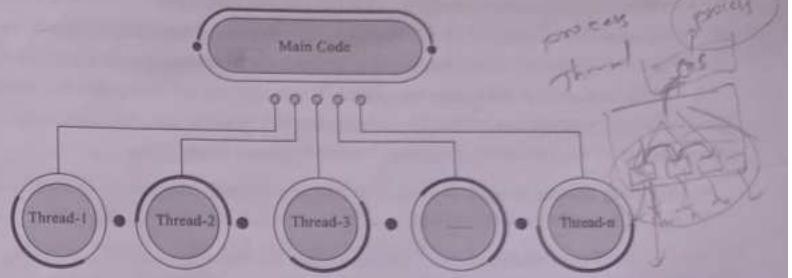


Fig 4.2 : Executing Multiple Threads

Multithreading refers to the ability of a program to execute multiple threads of execution concurrently. A thread is a lightweight sub-process that shares the same memory space as the main process and can run in parallel with other threads.

THREADS VERSUS PROCESSES

More than one thread can be implemented within the same process, most often executing concurrently and accessing/sharing the same resources, such as memory; separate processes do not do this. Threads in the same process share the latter's instructions (its code) and context (the values that its variables reference at any given moment).

The key difference between the two concepts is that a thread is typically a component of a process. Therefore, one process can include multiple threads, which can be executing simultaneously. Threads also usually allow for shared resources, such as memory and data, while it is fairly rare for processes to do so. In short, a thread is an independent component of computation that is similar to a process, but the threads within a process can share the address space, and hence the data, of that process.

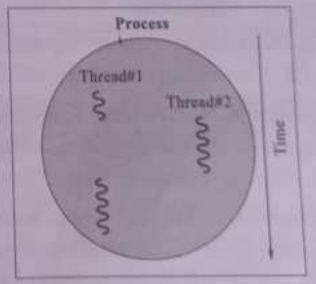


Fig 4.3 : A process with two threads of execution running on one processor

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Multithreading is very useful for saving time and improving performance, but it cannot be applied everywhere. In the previous FIFA example, the music thread is independent of the thread that takes your input and the thread that takes your input is independent of the thread that runs your opponent. These threads run independently because they are not inter-dependent. Therefore, multithreading can be used only when the dependency between individual threads does not exist.

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.

4.7 PROS AND CONS OF MULTITHREADING

Multithreading is a programming technique that allows multiple threads to run concurrently within a single process. In python, multithreading is often used to perform multiple tasks simultaneously, which can help improve the overall performance of a program. Here are some of the pros and cons of multithreading in python:

Pros:

1. **Improved performance:** Multithreading can improve the performance of a program, as multiple threads can run concurrently, utilizing multiple cores of the CPU.
2. **Increased responsiveness:** Multithreading can make a program more responsive by allowing it to perform multiple tasks simultaneously. This can help to improve the user experience of an application.
3. **Better resource utilization:** Multithreading can better utilize the available resources, such as CPU, memory, and I/O.
4. **Simplified programming:** Multithreading can simplify the programming of complex tasks, by dividing them into smaller, more manageable tasks that can run concurrently.
5. **Enhanced user experience:** Multithreading can enhance the user experience by making the application more responsive, as it can perform multiple tasks concurrently.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Cons:

1. **Complexity:** Multithreading can increase the complexity of a program, as it requires careful synchronization and coordination of the threads to avoid race conditions and deadlocks.
2. **Overhead:** Multithreading can introduce overhead, as the operating system has to switch between the threads, leading to increased context switching and memory usage.
3. **Debugging difficulties:** Multithreaded programs can be difficult to debug, as errors can be unpredictable and difficult to reproduce.
4. **Scalability:** Multithreading may not scale well as the number of threads increases, as the overhead of thread creation and management can become a bottleneck.
5. **Deadlocks and Race Conditions:** Multithreaded programs can suffer from deadlocks and race conditions, which can cause a program to become unresponsive or behave unpredictably.

4.8 CREATE THREADS USING THREADING MODULE

The **threading module** in python provides several methods for working with threads. Here are some of the commonly used methods:

- **threading.Thread(target, args=None, kwargs=None):** This method creates a new thread object. The target argument specifies the callable object to be invoked by the thread. The optional args and kwargs arguments can be used to pass additional arguments to the target function.
- **threading.current_thread():** This method returns a reference to the current Thread object.
- **threading.enumerate():** This method returns a list of all the currently running threads.
- **threading.active_count():**This method returns the number of Thread objects that are currently alive.

In addition to the methods, The **Thread class** in threading module provides several methods for working with threads. Here are some of the commonly used methods:

- **start():**This method starts the thread by calling the run() method of the thread. Starts the activity of a thread. It must be called only once for each thread because it will throw a runtime error if called multiple times. It returns immediately and the thread begins executing in the background.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

- **run():** This method is the entry point for the thread. It is called by the start() method and should be overridden in the subclass to define the thread's behaviour.
- **join(timeout=None):** This method waits for the thread to complete. If the optional timeout argument is specified, the method will block for at most the specified number of seconds.
- **is_alive():** This method returns True if the thread is still alive, otherwise it returns False.
- **getName() and setName(name):** These methods are used to get or set the name of the thread.
- **isDaemon() and setDaemon(daemonic):** These methods are used to get or set the daemon flag of the thread. A daemon thread is a thread that runs in the background and can be terminated at any time.

Creating thread using threading Module : The **threading module** provides a way to create and manage threads. To use threading, you create a new thread object and specify the function that should be run in the new thread. This function is called the **target function**. You can also pass arguments to the target function. Here are the steps to create a new thread using the threading module:

1. **Import the threading module:** The first step is to import the threading module using the import statement.

```
import threading
```

2. **Define a function to be run in the thread:** The next step is to define a function that will be executed in the new thread. This function should contain the code that you want to run concurrently with the main thread.

```
def my_function():
  # code to be executed in the new thread
```

3. **Create a new Thread object:** To create a new thread, you need to create a Thread object and pass the function you want to execute as an argument. You can also provide a name for the thread (optional).

```
my_thread = threading.Thread(target=my_function, name='my_thread')
```

4. **Start the new thread:** To start the new thread, you need to call the start() method on the Thread object.

```
my_thread.start()
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Example:

```
import time
import threading

def calc_square(numbers):
    print("Calculate square numbers")
    for n in numbers:
        time.sleep(1)
        print('square: ', n*n)

def calc_cube(numbers):
    print("Calculate cube of numbers")
    for n in numbers:
        time.sleep(1)
        print('cube: ', n*n*n)

arr = [2,3,4,5,6,7,8,9]

t = time.time()

t1= threading.Thread(target=calc_square, args=(arr,))
t2= threading.Thread(target=calc_cube, args=(arr,))

t1.start()
t2.start()

t1.join()
t2.join()
print("Done in : ",time.time()-t)
```

You can also create a new thread using the threading module in python by defining a class that extends the Thread class and overriding the __init__() and run() methods. Here is an example:

```
import threading
import time

# Define a new class that extends the Thread class
class MyThread(threading.Thread):
    def __init__(self, name, delay):
        threading.Thread.__init__(self)
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
        self.name = name
        self.delay = delay

    # Override the run method to define the behavior of the thread
    def run(self):
        print("Starting " + self.name)
        count = 0
        while count <= 5:
            time.sleep(self.delay)
            print(self.name + " count: " + str(count))
            count += 1
        print("Exiting " + self.name)

# Create two instances of the MyThread class
thread1 = MyThread("Thread-1", 1)
thread2 = MyThread("Thread-2", 2)

# Start the threads
thread1.start()
thread2.start()

# Wait for the threads to finish
thread1.join()
thread2.join()

print("Main thread exiting...")
```

Output :

```
StartingThread-1 StartingThread-2
Thread-1 count: 0
Thread-1 count: 1Thread-2 count: 0
Thread-1 count: 2
Thread-2 count: 1
Thread-1 count: 3
Thread-1 count: 4
Thread-2 count: 2
Thread-1 count: 5
Exiting Thread-1
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL


```
Thread-2 count: 3
Thread-2 count: 4
Thread-2 count: 5
Exiting Thread-2
Main thread exiting...
```

4.9 CREATE MULTIPLE THREADS WHICH PERFORM DIFFERENT TASKS

Multithreading allows you to break down an application into multiple sub-tasks and run these tasks simultaneously. If you use multithreading properly, your application speed, performance, and rendering can all be improved. Python supports constructs for both multiprocessing as well as multithreading. There are two main modules in python that can be used to handle threads: **threading** and **multiprocessing**.

Here is an example to check how long it takes for a code to perform two different tasks (finding square, cube of different numbers) with and without multithreading in python:

```
import time
def sqr(n):
    for x in n:
        time.sleep(1)
        print("Square of number",x,"is: ", x*x)
def cube(n):
    for x in n:
        time.sleep(1)
        print("Cube of number",x,"is: ", x*x*x)
n=[1,2,3,4,5,6,7,8]
start=time.time()
sqr(n)
cube(n)
end=time.time()
print("Total time taken for execution is:",end-start)
```

Output :

```
Square of number 1 is:  1
Square of number 2 is:  4
Square of number 3 is:  9
Square of number 4 is:  16
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
t2=MyThread()
t2.name = "CubeThread"
t1.start()
t2.start()
t1.join()
t2.join()
print("Main Thread Ends")
```

Output :

```
SquareThreadCubeThread — 11
SquareThread - 4
CubeThread - 8
CubeThreadSquareThread — 279
CubeThread - 64SquareThread - 16
CubeThread - 125
SquareThread - 25
CubeThread - 216
Square Thread - 36
Cube Thread - 343
Square Thread - 49
Cube Thread - 512
Square Thread - 64
Cube Thread SquareThread — 72981
Square Thread Cube Thread — 1001000
Main Thread Ends
```

4.10 THREADS USING START(), JOIN(), IS_ALIVE(), GETNAME(), SETNAME(), ACTIVE_COUNT() AND CURRENT_THREAD() METHODS

The **threading module** in python provides several methods for working with threads. Here are some of the commonly used methods:

- **threading.current_thread():** This method returns a reference to the current Thread object.
- **threading.active_count():**This method returns the number of Thread objects that are currently alive.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

In addition to the methods, The **Thread class** in threading module provides several methods for working with threads. Here are some of the commonly used methods:

- **start():** This method starts the thread by calling the run() method of the thread. Starts the activity of a thread. It must be called only once for each thread because it will throw a runtime error if called multiple times. It returns immediately and the thread begins executing in the background.
- **run():** This method is the entry point for the thread. It is called by the start() method and should be overridden in the subclass to define the thread's behaviour.
- **join(timeout=None):** This method waits for the thread to complete. If the optional timeout argument is specified, the method will block for at most the specified number of seconds.
- **is_alive():** This method returns True if the thread is still alive, otherwise it returns False.
- **getName() and setName(name) :** These methods are used to get or set the name of the thread.

1. start() and join() methods examples

In Python, the **join() method** is used to wait for a thread to complete its execution before continuing with the main thread. Here's an example that shows the difference between using join() and not using join():

Without join() method:

```
import threading
import time

def print_numbers():
    for i in range(5):
        print(i)
        time.sleep(1)

t = threading.Thread(target=print_numbers)
t.start()

print("Main thread continues to execute.")
```

In this example, we create a new thread t that calls the print_numbers() function. We start the thread using t.start() and then print "Main thread continues to execute.". Since

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

```
Square of number 5 is:  25
Square of number 6 is:  36
Square of number 7 is:  49
Square of number 8 is:  64
Cube of number 1 is:  1
Cube of number 2 is:  8
Cube of number 3 is:  27
Cube of number 4 is:  64
Cube of number 5 is:  125
Cube of number 6 is:  216
Cube of number 7 is:  343
Cube of number 8 is:  512
Total time taken for execution is: 16.398146629333496
```

The above is the output time taken to execute the program without using threads. Now let us use threads and see what happens to the same program:

```
import threading
import time
def sqr(n):
  for x in n:
    time.sleep(1)
    print("Square of number ",x,"is:",x*x)
def cube(n):
  for x in n:
    time.sleep(1)
    print("Cube of number ",x,"is:",x*x*x)
n=[1,2,3,4,5,6,7,8]
start=time.time()
t1=threading.Thread(target=sqr, args=(n,))
t2=threading.Thread(target=cube, args=(n,))
t1.start()
t2.start()
t1.join()
t2.join()
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL


```
end=time.time()
print("Total time taken for execution is:",end-start)
```

Output :

```
Cube of number Square of number  11  is: is:  11
Square of number Cube of number  22  is: is:  48
Square of number 3 is:  9
Cube of number 3 is:  27
Square of number 4 is:  16
Cube of number 4 is:  64
Square of number 5 is: Cube of number  255  is:  125
Square of number Cube of number  66  is: is:  36216
Cube of number Square of number  77  is: is:  34349
Square of number Cube of number  88  is: is:  64512
Total time taken for execution is: 8.289551258087158
```

EXAMPLE-2

```
from threading import *
from time import *

class MyThread(Thread):
  def __init__(self):
    Thread.__init__(self)
  def run(self):
    name=current_thread().name
    if name=="SquareThread":
      for i in range(1,11):
        print(name,"-",(i*i))
        sleep(1)
    elif name=="CubeThread":
      for i in range(1,11):
        print(name,"-",(i*i*i))
        sleep(1)
t1=MyThread()
t1.name = "SquareThread"
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

we don't use join(), the main thread will continue to execute immediately and will not wait for the t thread to complete its execution. As a result, the output may be interleaved and not sequential.

Output :

```
0Main thread continues to execute.
1
2
3
4
```

With join() method:

```
import threading
import time

def print_numbers():
    for i in range(5):
        print(i)
        time.sleep(1)
t = threading.Thread(target=print_numbers)
t.start()

print("Main thread waits for the child thread to complete.")
t.join()
print("Main thread continues to execute.")
```

In this example, we create a new thread t that calls the print_numbers() function. We start the thread using t.start() and then print "Main thread waits for the child thread to complete.". We then use t.join() to wait for the t thread to complete its execution before continuing with the main thread. Finally, we print "Main thread continues to execute.".

As a result, the output will be sequential and not interleaved, and the main thread will wait for the child thread to complete its execution before continuing with its own execution.

Output:

```
Main thread waits for the child thread to complete.0
1
2
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL


```
3
4
Main thread continues to execute.
```

2. is_alive() Example

In threading module, the is_alive() method is used to check whether a thread is currently running or not. It returns True if the thread is running, and False otherwise. Here's an example:

```
import threading
import time

def my_function():
    print("Starting my_function")
    time.sleep(2)
    print("Ending my_function")

my_thread = threading.Thread(target=my_function)
print("Is my_thread alive?", my_thread.is_alive())

my_thread.start()
print("Is my_thread alive?", my_thread.is_alive())
time.sleep(1)
print("Is my_thread alive?", my_thread.is_alive())

my_thread.join()
print("Is my_thread alive?", my_thread.is_alive())
```

The output of this program will be:

```
Is my_thread alive? False
Starting my_function
Is my_thread alive? True
Is my_thread alive? True
Ending my_function
Is my_thread alive? False
```

As you can see, my_thread.is_alive() returns False before the thread has started running, and True after it has started running. After the thread has finished running, my_thread.is_alive() returns False again. This shows that the is_alive() method can be used to check whether a thread is currently running or not.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

3. getName() and setName(name) Example

getName() method is used to return the thread's name, **setName(name)** method is used for setting the thread's name. The name is a string used for identification purposes only. Here's an example:

```
import threading

def print_name():
    print(f"Thread name: {threading.currentThread().getName()}")

thread = threading.Thread(target=print_name)

# Get the name of the thread
print(f"Initial thread name: {thread.getName()}")

# Set a new name for the thread
thread.setName("MyThread")

thread.start()
thread.join()
# Get the updated name of the thread
print(f"Final thread name: {thread.getName()}")
```

Output :

```
Initial thread name: Thread-1 (print_name)
Thread name: MyThread
Final thread name: MyThread
```

4. threading.active_count() Example

In threading module, the **active_count() method** is used to get the number of currently running threads in the program. Here's an example:

```
import threading
import time

def my_function():
    print("Starting my_function")
    time.sleep(2)
    print("Ending my_function")

my_thread1 = threading.Thread(target=my_function)
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL


```
my_thread2 = threading.Thread(target=my_function)

my_thread1.start()
my_thread2.start()

print("Number of active threads:", threading.active_count())

my_thread1.join()
my_thread2.join()

print("All threads have finished running")
```

The output of this program will be:

```
Starting my_function
Starting my_function
Number of active threads: 3
Ending my_function
Ending my_function
All threads have finished running
```

As you can see, threading.active_count() returns the number of currently running threads, which is 3 in this case (including the main thread). Once both threads have finished running, the program continues with the main thread and prints "All threads have finished running".

5. threading.current_thread() Example

In threading module, the **current_thread()** method is used to get the current thread object. This can be useful when you want to know which thread is currently running a certain piece of code. Here's an example:

```
import threading

def my_function():
    print("Running in thread:", threading.current_thread().name)

my_thread = threading.Thread(target=my_function)
my_thread.start()
my_thread.join()

print("Running in thread:", threading.current_thread().name)
```

In this example, we define a function my_function() that prints the name of the current

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

thread using threading.current_thread().name. We create a new thread to run this function and start it with my_thread.start(). After starting the thread, we print the name of the current thread again using threading.current_thread().name. This will print the name of the main thread.

The output of this program will be:

```
Running in thread: Thread-1 (my_function)
Running in thread: MainThread
```

4.11 THREAD SYNCHRONIZATION IN MULTITHREADED ENVIRONMENT

To deal with race conditions, deadlocks, and other thread-based issues, the threading module provides the **Lock** object. The idea is that when a thread wants access to a specific resource, it acquires a lock for that resource. Once a thread locks a particular resource, no other thread can access it until the lock is released. As a result, the changes to the resource will be atomic, and race conditions will be handled.

A lock is a low-level synchronization primitive implemented by the threading module. Lock object provides a way to synchronize access to shared resources by allowing only one thread to acquire the lock at a time. At any given time, a lock can be in one of 2 states: **locked** or **unlocked**. It supports two methods:

1. **acquire()**

When the lock-state is unlocked, calling the acquire() method will change the state to locked and return. However, If the state is locked, the call to acquire() is blocked until the release() method is called by some other thread.

2. **release()**

The release() method is used to set the state to unlocked, i.e., to release a lock. It can be called by any thread, not necessarily the one that acquired the lock.

Here's an example of using the acquire() and release() methods of the Lock object:

```
import threading

balance = 0  # Shared resource
lock = threading.Lock()  # Create a lock

def deposit(amount):
    global balance
    lock.acquire()  # Acquire the lock
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL


```
    try:
        balance += amount
    finally:
        lock.release()  # Release the lock

def withdraw(amount):
    global balance
    lock.acquire()  # Acquire the lock
    try:
        if balance >= amount:
            balance -= amount
        else:
            print("Insufficient balance")
    finally:
        lock.release()  # Release the lock

thread1 = threading.Thread(target=deposit, args=(1000,))
thread2 = threading.Thread(target=withdraw, args=(500,))
thread3 = threading.Thread(target=withdraw, args=(1500,))

thread1.start()
thread2.start()
thread3.start()

thread1.join()
thread2.join()
thread3.join()

print(f"Final balance: {balance}")
```

Output :

```
Insufficient balance
Final balance: 500
```

In this example, we define two functions deposit() and withdraw() that modify the balance variable. We create a Lock object named lock to synchronize access to the balance variable. By using the acquire() and release() methods of the Lock object, we ensure that only one thread can access the balance variable at a time, thus preventing race conditions and ensuring the correctness of the final balance.

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

Here is an example program that demonstrates the need for thread synchronization in Python:

Without synchronization

Here's an example of a simple program that uses a shared counter variable to count the number of times a function is called, without synchronization:

```
import threading
counter = 0

def increment_counter():
    global counter
    counter += 1
    print(f"Counter value: {counter}")

threads = []
for i in range(10):
    t = threading.Thread(target=increment_counter)
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

In this example, we define a global variable counter and a function increment_counter() that increments the counter and prints its value. We create 10 threads and start them, each calling the increment_counter() function. The expected result is that the counter is incremented by 10 each time the program is run.

However, without synchronization, the threads may access the counter variable concurrently and cause a race condition. The output of the program can be unpredictable and may vary each time it is run. Here's an example of the output:

```
Counter value: 1Counter value: 4Counter value: 2Counter value: 5Counter value: 6Counter value: 7Counter value: 8Counter value: 9
Counter value: 10
Counter value: 3
```

Now let's modify the program to use a Lock object to synchronize access to the counter variable:

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

With synchronization using Lock object

```
import threading

counter = 0
lock = threading.Lock()  # Create a lock

def increment_counter():
    global counter
    lock.acquire()  # Acquire the lock
    try:
        counter += 1
        print(f"Counter value: {counter}")
    finally:
        lock.release()  # Release the lock

threads = []
for i in range(10):
    t = threading.Thread(target=increment_counter)
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

In this modified program, we create a Lock object named lock and use the lock.acquire() and lock.release() methods to synchronize access to the counter variable. Now when we run the program, we get the expected output:

```
Counter value: 1
Counter value: 2
Counter value: 3
Counter value: 4
Counter value: 5
Counter value: 6
Counter value: 7
Counter value: 8
Counter value: 9
Counter value: 10
```

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL

As we can see, the counter value is incremented by 10, and each value is unique. By using synchronization, we ensure that the threads access the shared counter variable one at a time, thus preventing race conditions and ensuring the correctness of the counter value.

Apart from locks, python also supports some other mechanisms to handle thread synchronization as listed below :

1. RLocks
2. Queues
3. Semaphores
4. Conditions
5. Events
6. Barriers

WARNING : XEROX / PHOTOCOPYING OF THIS BOOK IS ILLEGAL