MODULE IV

Object Oriented Programming

- **Design with classes –**

  Objects and Classes Methods

  Instance variables

  Constructor

  Accessor and Mutator
- **Data-Modelling Examples**
- **Structuring classes with inheritance and polymorphism.**
- **Abstract classes**
- **Interfaces**
- **Exceptions**

  Handle a single exception Handling multiple

  exceptions

Programming languages that allow the programmer to define new classes of objects are called **object-oriented languages.**

OBJECTS AND CLASSES

- The interface or set of methods that can be used with a class of objects
- The attributes of an object that describe its state from the user's point of view
- How to instantiate a class to obtain an object

A class definition is like a blueprint for each of the objects of that class.

contain
s

- • Definitions of all of the methods
- • Descriptions of the data structures used to maintain the state of an object.

An object packages a set of data values, its state, and a set of operations, its methods in a single entity that can be referenced with a name.

```
class <class name>(<parent class name>):
    <method definition-1>
    ...
    <method definition-n>
```

The class definition syntax has two parts:

1. Class header
2. A set of method definitions that follow the class header.

The class header consists of the class name and the parent class name.

The class name is a Python identifier.

The parent class name refers to another class.

All Python classes are organized in a tree-like class hierarchy.

At the top, or root, of this tree is the most abstract class, named **object**.

Each class immediately below another class in the hierarchy is referred to as a subclass, whereas the class immediately above it, if there is one, is called its parent class.

If the parenthesized parent class name is omitted from the class definition, the new class is automatically made a subclass of object

```python
"""
File: student.py
Resources to manage a student's name and test scores.
"""

class Student(object):
    """Represents a student."""
```

```python
def  __init__(self, name, number):
     """Constructor creates a Student with the given
     name and number of scores and sets all scores
     to 0."""
     self.name = name
     self.scores = []
     for count in range(number):
         self.scores.append(0)

def getName(self):
    """Returns the student's name."""
    return self.name

def setScore(self, i, score):
    """Resets the ith score, counting from 1."""
    self.scores[i - 1] = score
```

```python
def getScore(self, i):
    """Returns the ith score, counting from 1."""
    return self.scores[i - 1]

def getAverage(self):
    """Returns the average score."""
    return sum(self.scores) / len(self.scores)

def getHighScore(self):
    """Returns the highest score."""
    return max(self.scores)

def __str__(self):
    """Returns the string representation of the
    student."""
    return "Name: " + self.name + "\nScores: " + \
           " ".join(map(str, self.scores))
```

```
>>> from student import Student
>>> s = Student("Maria", 5)
>>> print(s)
Name: Maria
Scores: 0 0 0 0 0
>>> s.setScore(1, 100)
>>> print(s)
Name: Maria
Scores: 100 0 0 0 0
>>> s.getHighScore()
100
>>> s.getAverage()
20
>>> s.getScore(1)
100
>>> s.getName()
'Maria'
```

Docstrings

```
help(Student)
```

Method Definitions

Method definition must include a first parameter named self.

The interpreter binds the parameter self to that object.

s.getScore(4)

binds the parameter self in the method getScore to the Student object referenced by the variable S.

A method automatically returns the value None when it includes no return statement

THE __INIT__ METHOD AND INSTANCE VARIABLES

__init__ must begin and end with two consecutive underscores.

This method is also called the class's constructor, it will run automatically when a user instantiates the class.

The purpose of the constructor is to initialize an individual object's attributes.

The attributes of an object are represented as **instance variables.**

These variables serve as storage for its state.

The scope of an instance variable is the entire class definition. The object

lifetime of an instance variable is the lifetime of the enclosing

The __str__ Method

Builds and returns a string representation of an object's state.

When the str function is called with an object, that object's __str__ method is automatically invoked to obtain the string that str returns

Accessors and Mutators

Methods that allow a user to observe but not change the state of an object are called **accessors.**

Methods that allow a user to modify an object's state are called **mutators**

AN OBJECT COMES INTO BEING WHEN ITS CLASS IS INSTANTIATED.

Python virtual machine will eventually recycle its storage during a process called **garbage collection.**

DATA-MODELLING EXAMPLES

Rational Numbers

A rational number consists of two integer parts, a numerator and a denominator.

Format *numerator / denominator.*

Python has no built-in type for rational numbers.

```
>>> oneHalf = Rational(1, 2)
>>> oneSixth = Rational(1, 6)
>>> print(oneHalf)
1/2
>>> print(oneHalf + oneSixth)
2/3
>>> oneHalf == oneSixth
False
>>> oneHalf > oneSixth
True
```

RATIONAL NUMBER ARITHMETIC AND OPERATOR OVERLOADING

Python allows to overload many of the built-in operators for use with new data types.

Operator	Method Name
+	__add__
-	__sub__
*	__mul__
/	__div__
%	__mod__

The object on which the method is called corresponds to the left operand.

The method's second parameter corresponds to the right operand.

The code x + y is actually shorthand for the code x.__add__(y).

```python
"""
File: rational.py
Resources to manipulate rational numbers.
"""
class Rational(object):
 """Represents a rational number."""
 def __init__(self, numer, denom) :
  """Constructor creates a number with the given
  numerator and denominator and reduces it to lowest
  terms."""
  self.numer = numer
  self.denom = denom
  self._reduce()
 def numerator(self):
  """Returns the numerator."""
  return self.numer
 def denominator(self):
  """Returns the denominator."""
  return self.denom
 def __str__(self):
  """Returns the string representation of the
  number."""
  return str(self.numer) + "/" + str(self.denom)
 def _reduce(self):
  """Helper to reduce the number to lowest terms."""
  divisor = self._gcd(self.numer, self.denom)
  self.numer = self.numer // divisor
  self.denom = self.denom // divisor
```

```
 def _gcd(self, a, b):
  """Euclid's algorithm for greatest common
  divisor (hacker's version)."""
  (a, b) = (max(a, b), min(a, b))
  while b > 0:
   (a, b) = (b, a % b)
  return a
 def __add__(self, other):
  """Returns the sum of the numbers.
  self is the left operand and other is
  the right operand."""
  newNumer = self.numer * other.denom + \
  other.numer * self.denom
  newDenom = self.denom * other.denom
  print(Rational(newNumer, newDenom))

 def __sub__(self, other):
      newNumer=self.numer * other.denom - \
      other.numer * self.denom
      newDenom=self.denom * other.denom
      return Rational(newNumer, newDenom)
      print(Rational(newNumer,newDenom))

x=Rational(5,4)
y=Rational(6,4)
c=x-y
print(c)
c=x+y
print(c)
```

TO OVERLOAD AN ARITHMETIC OPERATOR, DEFINE A NEW METHOD USING THE APPROPRIATE METHOD NAME.

Type of Operation	Rule
Addition	$n_1/d_1 + n_2/d_2 = (n_1d_2 + n_2d_1) / d_1d_2$
Subtraction	$n_1/d_1 - n_2/d_2 = (n_1d_2 - n_2d_1) / d_1d_2$
Multiplication	$n_1/d_1 * n_2/d_2 = n_1n_2 / d_1d_2$

```
def __add__(self, other):
    """Returns the sum of the numbers.
    self is the left operand and other is
    the right operand."""
    newNumer = self.numer * other.denom + \
               other.numer * self.denom
    newDenom = self.denom * other.denom
    return Rational(newNumer, newDenom)
```

The code x + y is actually shorthand for the code x.__add__(y).

Operator	Meaning	Method
==	Equals	__eq__
!=	Not equals	__ne__
<	Less than	__lt__
<=	Less than or equal	__le__
>	Greater than	__gt__
>=	Greater than or equal	__ge__

Table 9-5 The comparison operators and methods

```python
def __lt__(self, other):
    """Compares two rational numbers, self and other,
    using <."""
    extremes = self.numer * other.denom
    means = other.numer * self.denom
    return extremes < means
```

```python
def __eq__(self, other):
    """Tests self and other for equality."""
    if self is other:                       # Object identity?
        return True
    elif type(self) != type(other):        # Types match?
        return False
    else:
        return self.numer == other.numer and \
               self.denom == other.denom
```

Structuring classes with inheritance and polymorphism

1. Data encapsulation. Restricting the manipulation of an object's state by external users to a set of method calls.

2. Inheritance. Allowing a class to automatically reuse and extend the code of similar but more general classes.

3. Polymorphism. Allowing several different classes to use the same general method names

In Python, all classes automatically extend the built-in object class, which is the most general class.

Inheritance Hierarchies

Objects in the natural world can be classified using **inheritance hierarchies**

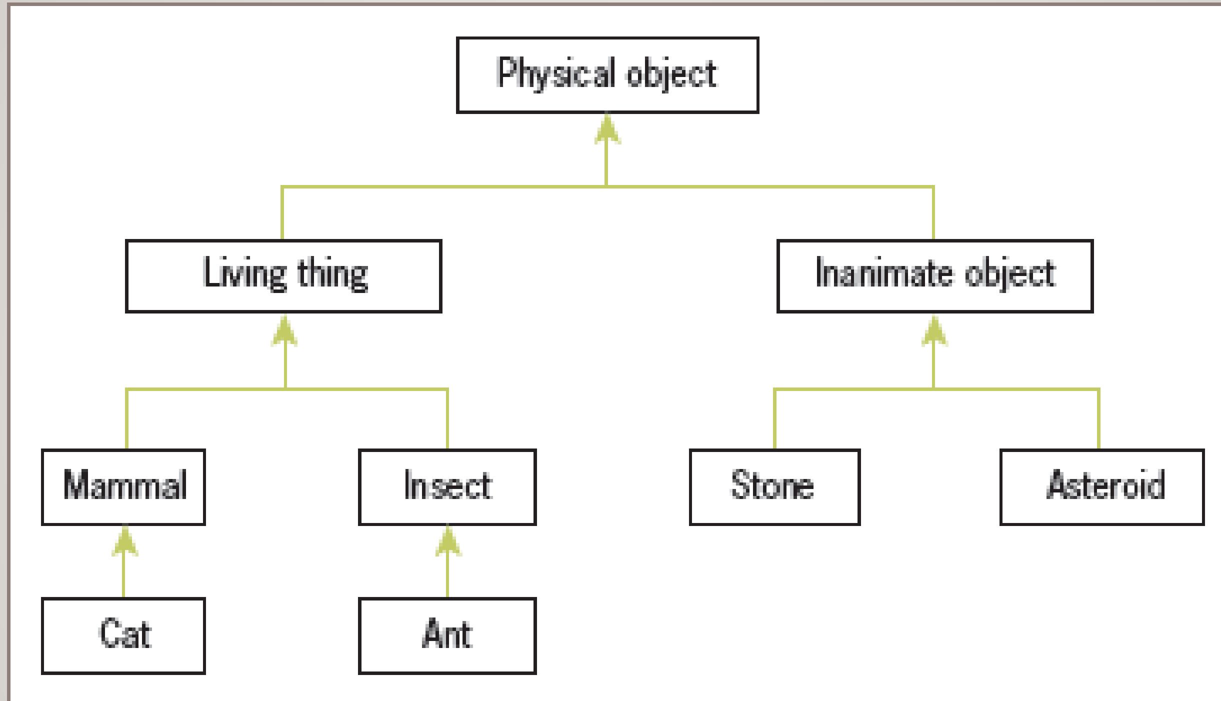

Figure 9-5 A simplified hierarchy of objects in the natural world

AT THE TOP OF A HIERARCHY IS THE MOST GENERAL CLASS OF OBJECTS.

This class defines features that are common to every object in the hierarchy.

For example, every physical object has a mass.

Classes just below this one have these features as well as additional ones.

Thus, a living thing has a mass and can also grow and die.

The path from a given class back up to the topmost one goes through all of that given class's ancestors.

Each class below the topmost one inherits attributes and behaviours from its ancestors and extends these with additional attributes and behaviour.

There are five types of inheritances:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Single Inheritance

This type of inheritance enables a subclass or derived class to inherit properties and characteristics of the parent class, this avoids duplication of code and improves code reusability.

```
#parent class
class Above:
    i = 5
    def fun1(self):
        print("Hey there, you are in the parent class")

#subclass
class Below(Above):
    i=10
    def fun2(self):
        print("Hey there, you are in the sub class")

temp1=Below()
temp2=Above()
temp1.fun1()
temp1.fun2()
temp2.fun1()
print(temp1.i)
print(temp2.i)
#temp2.fun2()
```

Multiple Inheritance

This inheritance enables a child class to inherit from more than one parent class.

This type of inheritance is not supported by java classes, but python does support this kind of inheritance.

It has a massive advantage that, gathering multiple characteristics from different classes.

```python
#parent class 1
class A:
    demo1=0
    def fun1(self):
        print("Printing C's value in A'S Function",self.demo1)

#parent class 2
class B:
    demo2=0
    def fun2(self):
        print("Printing C's value in B'S Function",self.demo2)

#child class
class C(A, B):
    def fun3(self):
        print("Hey there, you are in the child class")

# Main code
c = C()
c.demo1 = 10
c.demo2 = 5
c.fun1()
c.fun2()
c.fun3()
print("first number is : ",c.demo1)
print("second number is : ",c.demo2)
```

MULTILEVEL INHERITANCE

IN MULTILEVEL INHERITANCE, THE TRANSFER OF THE PROPERTIES OF CHARACTERISTICS IS DONE TO MORE THAN ONE CL... R TO GRA... O LEAF IN... .

```python
#parent class 1
class vehicle:
    def functioning(self):
        print("vehicles are used for transportation")

#child class 1
class car(vehicle):
    def wheels(self):
        print("a car has 4 wheels")

#child class 2
class electric_car(car):
    def speciality(self):
        print("electric car runs on electricity")

electric=electric_car()
electric.speciality()
electric.wheels()
electric.functioning()
```

Hierarchical Inheritance

This inheritance allows a class to host as a parent class for more than one child class or subclass.

This provides a benefit of sharing the functioning of methods with multiple child classes, hence avoiding code duplication.

```python
#parent class
class Parent:
    def parent_fun(self):
        print("Hey there, you are in the parent class")

#child class 1
class child1(Parent):
    def fun1(self):
        print("Hey there, you are in the child class 1")

#child class 2
class child2(Parent):
    def fun2(self):
        print("Hey there, you are in the child class 2")

#child class 3
class child3(Parent):
    def fun3(self):
        print("Hey there, you are in the child class 3")

# main program
child_obj1 = child1()
child_obj2 = child2()
child_obj3 = child3()
child_obj1.parent_fun()
child_obj1.fun1()
child_obj2.parent_fun()
```

HYBRID INHERITANCE

An inheritance is said hybrid inheritance if more than one type of inheritance is implemented in the same code.

This feature enables the user to utilize the feature of inheritance at its best.

This satisfies the requirement of implementing a code that needs multiple inheritances in implementation.

```python
class A:
 def fun1(self):
  print("Hey there, you are in class A")
class B(A):
 def fun2(self):
  print("Hey there, you are in class B")
class C(A):
 def fun3(self):
  print("Hey there, you are in class C")
class D(C,B):
 def fun4(self):
  print("Hey there, you are in the class D")
ref = D()
ref.fun4()
ref.fun3()
ref.fun2()
ref.fun1()
```

POLYMORPHIC METHODS

A subclass adds something extra, such as a new method or a data attribute, to the ensemble provided by its super class.

Two classes have the same interface, or set of methods available to external users.

One or more methods in a subclass override the definitions of the same methods in the super class to provide specialized versions of the abstract behaviour.

This methods are known as **polymorphic methods.**

The __str__method is an example for a polymorphic method

THE WORD 'POLY' MEANS MANY AND 'MORPHS' MEANS FORMS.

The process of representing "one form in many forms" is called a polymorphism.

Types of Polymorphism in Python:

The following are the examples, or implementations, of polymorphism:

1. Duck Typing Philosophy of Python
2. Overloading
    - Operator Overloading
    - Method Overloading
    - Constructor Overloading
3. Overriding
    - Method overriding
    - Constructor overriding

Duck Typing Philosophy of Python

Duck typing refers to the programming style, in which the object passed to a method supports all the attributes expected from it, at the runtime.

```
class Duck:
    def talk(self):
        print("Quack.. Quack")
class Dog:
    def talk(self):
        print("Bow...Bow")
class Cat:
    def talk(self):
        print("Moew...Moew ")


def m(obj):
    obj.talk()

duck = Duck()
m(duck)

cat = Cat()
m(cat)

dog = Dog()
m(dog)
```

```
Quack.. Quack
Moew...Moew
Bow...Bow
```

OVERLOADING IN PYTHON

1. **Operator Overloading**
2. **Method Overloading**
3. **Constructor Overloading**

Method overloading in Python:

If 2 methods have the same name but different types of arguments, then those methods are said to be overloaded methods.

```
class Demo:
    def m1(self):
        print('no-arg method')
    def m1(self, a):
        print('one-arg method')
    def m1(self, a, b):
        print('two-arg method')

d= Demo()
#d.m1()
#d.m1(10)
d.m1(10,20)
```

Output: `two-arg method`

Operator Overloading

+	→ object.__add__(self, other)
-	→ object.__sub__(self, other)
*	→ object.__mul__(self, other)
/	→ object.__div__(self, other)
//	→ object.__floordiv__(self, other)
%	→ object.__mod__(self, other)
**	→ object.__pow__(self, other)
+=	→ object.__iadd__(self, other)
-=	→ object.__isub__(self, other)
*=	→ object.__imul__(self, other)
/=	→ object.__idiv__(self, other)
//=	→ object.__ifloordiv__(self, other)
%=	→ object.__imod__(self, other)
**=	→ object.__ipow__(self, other)
<	→ object.__lt__(self, other)
<=	→ object.__le__(self, other)
>	→ object.__gt__(self, other)
>=	→ object.__ge__(self, other)
==	→ object.__eq__(self, other)
!=	→ object.__ne__(self, other)

CONSTRUCTOR OVERLOADING IN PYTHON

```
class Demo:
    def __init__(self):
        print('No-Arg Constructor')
    def __init__(self, a):
        print('One-Arg constructor')
    def __init__(self, a, b):
        print('Two-Arg constructor')


#d1=Demo()
#d1=Demo(10)
d1=Demo(10,20)
```

METHOD OVERRIDING IN PYTHON:

```
class P:
    def properties_status(self):
        print('Money, Land, Gold')
    def to_marry(self):
        print('Aravind')
class C(P):
    def study_status(self):
        print("Studies done waiting for job")
    def to_marry(self):
        print('Megha')

c=C()
c.properties_status()
c.to_marry()
c.study_status()
```

```
Money, Land, Gold
Megha
Studies done waiting for job
>>>
```

If child class does not have constructor, then parent class constructor will be executed at the time of child class object creation.

If child class has a constructor, then child class constructor will be executed at the time of child class object creation.

From child class constructor parent class constructor can be invoked by using super() method

Constructor Overriding in Python:

```
class Person:
    def __init__(self, name, age):
        self.name=name
        self.age=age
class Employee(Person):
    def __init__(self, name, age, eno, esal):
        super().__init__(name, age)
        self.eno=eno
        self.esal=esal
    def display(self):
        print('Employee Name:', self.name)
        print('Employee Age:', self.age)
        print('Employee Number:', self.eno)
        print('Employee Salary:', self.esal)

e1=Employee('Surabhi', 16, 872425,26000)
e1.display()
e2=Employee('Ranjith',20,872426,36000)
e2.display()
```

```
Employee Name: Surabhi
Employee Age: 16
Employee Number: 872425
Employee Salary: 26000
Employee Name: Ranjith
Employee Age: 20
Employee Number: 872426
Employee Salary: 36000
```

ABSTRACT CLASSES

- An abstract class is a class that cannot be instantiated.
- Can create classes that inherit from an abstract class.
- An abstract method is a method without an implementation.
- An abstract class may or may not include abstract methods.
- To define an abstract class, use the abc (abstract base class) module.

```
from abc import ABC


class AbstractClassName(ABC):
    pass
```

```python
from abc import *
class Demo1(ABC):
    @abstractmethod
    def m1(self):
        pass
    @abstractmethod
    def m2(self):
        pass
    def m3(self):
        print("Implemented method")
```

Program: Abstract methods

```python
from abc import ABC, abstractmethod

#Abstract Class
class Bank(ABC):
    def bank_info(self):
        print("Welcome to bank")
    @abstractmethod
    def interest(self):
        "Abstarct Method"
        pass
#Sub class/ child class of abstract class
class SBI(Bank):

    def interest(self):
        "Implementation of abstract method"
        print("In sbi bank 5 rupees interest")

s= SBI()
s.bank_info ()
s.interest()
```

Program: Abstract Class

```
Welcome to bank
In sbi bank 5 rupees interest
```

```python
from abc import ABC, abstractmethod

#Abstract Class
class Bank(ABC):
    def bank_info(self):
        print("Welcome to bank")
    @abstractmethod
    def interest(self):
        "Abstarct Method"
        pass
#Sub class/ child class of abstract class
class SBI(Bank):

    def balance(self):
        print("Balance is 100")
s= SBI()
s.bank_info ()
s.balance()
```

PROGRAM:ABSTRACT CLASS

Output: TypeError: Can't instantiate abstract class SBI with abstract methods interest

PROGRAM: ABSTRACT CLASS AND SUBCLASS OF ABSTRACT CLASS

```python
from abc import ABC, abstractmethod

#Abstract Class
class Bank(ABC):
    def bank_info(self):
        print("Welcome to bank")
    @abstractmethod
    def interest(self):
        "Abstarct Method"
        pass
#Sub class/ child class of abstract class
class SBI(Bank):

    def balance(self):
        print("Balance is 100")

class Sub1(SBI):
    def interest(self):
        print("In sbi bank interest is 5 rupees")

s= Sub1()
s.bank_info ()
s.balance()
s.interest()
```

```
Welcome to bank
Balance is 100
In sbi bank interest is 5 rupees
```

Program: Abstract class can also contain concrete methods

```python
from abc import ABC, abstractmethod

#Abstract Class
class Bank(ABC):
    def bank_info(self):
        print("Welcome to bank")
    @abstractmethod
    def interest(self):
        "Abstarct Method"
        pass
    def offers(self):
        print("Providing offers")

#Sub class/ child class of abstract class
class SBI(Bank):
    def interest(self):
        print("In SBI bank 5 rupees interest")


s= SBI()
s.bank_info ()
s.interest()
```

Output:

```
Welcome to bank
In SBI bank 5 rupees interest
```

- **Every abstract class in Python should be derived from the ABC class which is present in the abc module.**

- **Abstract class can contain Constructors, Variables, abstract methods, non-abstract methods, and Subclass.**

- **Abstract methods should be implemented in the subclass or child class of the abstract class.**

•If in subclass the implementation of the abstract method is not provided, then that subclass, automatically, will become an abstract class.

•Then, if any class is inheriting this subclass, then that subclass should provide the implementation for abstract methods.

•Object creation is not possible for abstract class.

•Can create objects for child classes of abstract classes to access implemented methods.

INTERFACES

An interface is an abstract class which can contains only abstract methods.

In python there is no separate keyword to create an interface.

Create interfaces by using abstract classes which have only abstract methods.

Interface can contain:

- Constructors
- Variables
- Abstract methods
- Sub class

An interface acts as a template for designing classes.

Abstract methods are those methods without implementation or which are without the body.

So the interface just defines the abstract method without implementation.

The implementation of these abstract methods is defined by classes that implement an interface.

Program: Interface having two abstract methods and one sub class

```
from abc import ABC, abstractmethod
class Bank(ABC):
   @abstractmethod
   def balance_check(self):
       pass
   @abstractmethod
   def interest(self):
       pass

class SBI(Bank):
   def balance_check(self):
       print("Balance is 100 rupees")
   def interest(self):
       print("SBI interest is 5 rupees")


s = SBI()
s.balance_check()
s.interest()
```

Output:

```
Balance is 100 rupees
SBI interest is 5 rupees
```

EXCEPTIONS

Types of Error:
In any programming language there are 2 types of errors possible:

- **Syntax Errors**
- **Runtime Errors**

Syntax Errors
The errors which occur because of invalid syntax are called syntax errors.

RUNTIME ERRORS

While executing the program if something goes wrong then we will get Runtime Errors.

They might be caused due to,

1. **End user input**
2. **Programming logic**
3. **Memory problems etc.**

Such types of errors are called exceptions.

Runtime Error example

```
print(10/0)
```

What is an Exception

An unwanted or unexpected event which disturbs the normal flow of the program is called exception.

Whenever an exception occurs, then immediately program will terminate abnormally.

In order to get program executed normally, handle those exceptions on high priority.

Exception Handling

Exception handling is the process, in which define a way, so that the program doesn't terminate abnormally due to the exceptions.

In python, for every exception type, a corresponding class is available and every exception is an object to its corresponding class.

Whenever an exception occurs, Python Virtual Machine (PVM) will create the corresponding exception object and will check for handling code.

If handling code is not available, then Python interpreter terminates the program abnormally and prints corresponding exception information to the console.

The rest of the program won't be executed.

Exception Hierarchy

Every Exception in Python is a class.

The BaseException class is the root class for all exception classes in python exception hierarchy and all the exception classes are child classes of BaseException.

How to Handle Exception in Python?

Using try except statements

TRY BLOCK:

try is a keyword in python.

The code which may or expected to raise an exception, should be written inside the try block.

except block:

except is a keyword in python.

The corresponding handling code for the exception, if occurred, needs to be written inside the except block.

if the code in the try block raises an exception, then only execution flow goes to the except block for handling code.

If there is no exception raised by the code in the try block, then execution flow won't go to the except block.

Code with exception

```
print('One')
print('Two')

try:
    print(10/0)
except ZeroDivisionError:
    print("Exception passed")
print('Four')
print('Five')
```

```
One
Two
Exception passed
Four
Five
>>> 
```

Code without exception

```
print('One')
print('Two')

try:
    print(10/2)
    print("No Exception")
except ZeroDivisionError:
    print("Exception passed")
print('Four')
print('Five')
```

```
>>>
===============
One
Two
5.0
No Exception
Four
Five
>>> 
```

PRINTING EXCEPTION INFORMATION IN PYTHON:

Exception information can be accessed by creating a reference to the exception.

```
try:
   print(10/0)
except ZeroDivisionError as z:
   print("Exception information:", z)
```

```
>>>
================== RESTART: D:/Programs/py
Exception information: division by zero
>>> |
```

TRY WITH MULTIPLE EXCEPT BLOCKS IN PYTHON:

try with multiple except blocks are allowed in python.

There may be the possibility that a piece of code can raise different exceptions in different cases.

For every exception type a separate except block has to write:

```
try:
   x=int(input("Enter First Number: "))
   y=int(input("Enter Second Number: "))
   print(x/y)
except ZeroDivisionError:
   print("Can't Divide with Zero")
except ValueError:
   print("please provide int value only")
```

```
Enter First Number: 4
Enter Second Number: 0
Can't Divide with Zero
>>>
================== RESTART: D:/
Enter First Number: 6
Enter Second Number: o
please provide int value only
>>>
```

1

2

ONE EXCEPT BLOCK – MULTIPLE EXCEPTIONS

except (Exception1, Exception2, exception3,..):

or

except (Exception1, Exception2, exception3,..) as msg:

```
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
except (ZeroDivisionError,ValueError) as e:
    print("Please Provide valid numbers only and problem is: ", e)
```

```
Enter First Number: 3
Enter Second Number: 0
Please Provide valid numbers only and problem is:  division by zero
>>>
================= RESTART: D:/Programs/python/exception1.py =================
Enter First Number: 5
Enter Second Number: o
Please Provide valid numbers only and problem is:  invalid literal for int() wit
h base 10: 'o'
>>>
```

DEFAULT EXCEPT BLOCK

- Default except block is used to handle any type of exceptions.
- It's not required to mention any exception type for the default block.
- If there is no idea of what expectation the code could raise, then go for default except block.

```
try:
   x=int(input("Enter First Number: "))
   y=int(input("Enter Second Number: "))
   print(x/y)
except ZeroDivisionError:
   print("ZeroDivisionError: Can't divide with zero")
except:
   print("Default Except: Please provide valid input only")
```

finally Block

finally is a keyword in python.

Used to write a finally block to do clean-up activities.

```python
try:
    print("try block")
except:
    print("except block")
finally:
    print("finally block")
```

Output:

```
try block
finally block
```

```python
try:
    print("try block")
    print(10/0)
except ZeroDivisionError:
    print("except block")
finally:
    print("finally block")
```

Output:

```
try block
except block
finally block
```

end