Unit-5 :: Object Oriented Programming

Introduction to Classes and Objects

Classes and objects are the two basic concepts in object oriented programming. An object is an instance of a class. A class creates a new type and provides a blueprint or a template used for creating objects. In Python, everything is an object of some class. In Python, the base class for all the classes is ***object***.

Creating or Defining a Class

The syntax for defining a class is as follows:

```
class  class_name:
        <statement_1>
        <statement_2>
        ...
        ...
        <statement_n>
```

The statements in the syntax can be variables, control statements, or function definitions. The variables in a class are known as ***class variables***. The functions defined in a class are known as ***class methods***. Class variables and class methods together are called as ***class members***.

The class methods can access all class variables. The class members can be accessed outside the class by using the object of that class.

Creating Objects

Once a class is defined, we can create objects on it. Syntax for creating an object is as follows:

```
object_name = class_name( )
```

Creating an object for a class is known as ***instantiation***. We can access the class members using an object along with the dot (.) operator as follows:

```
object_name.variable_name
object_name.method_name(args_list)
```

Following code creates a Student class and one object ***s*** which is used to access the members of the Student class:

```
class  Student:
        #branch is a class variable
        branch = "CSE"
        def  read_student_details(self, rno, n):
                #regdno and name are instance variables
                self.regdno = rno
                self.name = n
        def  print_student_details(self):
                print("Student regd.no. is", self.regdno)
                print("Student name is", self.name)
                print("Student branch is", Student.branch)
s = Student( )  #s is an object of class Student
s.read_student_details("16PA1A0501", "Ramesh")
s.print_student_details( )
```

Class Variables and Instance Variables

Variables which are created inside a class and outside methods are known as class variables. Class variables are common for all objects. So, all objects share the same value for each class variable. Class variables are accessed as ***ClassName.variable_name***.

Variables created inside the class methods and accessed using the ***self*** variable are known as instance variables. Instance variable value is unique to each object. So, the values of instance variables differ from one object to the other. Consider the following class:

```
class  Student:
        #branch is a class variable
        branch = "CSE"
        def  read_student_details(self, rno, n):
                #regdno and name are instance variables
                self.regdno = rno
                self.name = n
        def  print_student_details(self):
                print("Student regd.no. is", self.regdno)
                print("Student name is", self.name)
                print("Student branch is", Student.branch)
```

In the above *Student* class, *branch* is a class variable and *regdno* and *name* are instance variables.

Public and Private Instance Variables

By default, instance variables are public. They are accessible throughout the class and also outside the class. An instance variable can be made private by using _ _ (double underscore) before the variable name. Private instance variables can be accessed only inside the class and not outside the class. Following example demonstrates private instance variable:

```
class Student:
        #branch is a class variable
        branch = "CSE"
        def  read_student_details(self, rno, n):
                #regdno and name are instance variables
                self.__regdno = rno #regdno is private variable
                self.name = n
        def  print_student_details(self):
                print("Student regd.no. is", self.__regdno)
                print("Student name is", self.name)
                print("Student branch is", Student.branch)
s = Student()
s.read_student_details("16PA1A0501", "Ramesh")
s.print_student_details()
print(s.__regdno)  #This line gives error as ___regdno is private.
```

self Argument

The ***self*** argument is used to refer the current object (like ***this*** keyword in Java). Generally, all class methods by default should have at least one argument which should be the ***self*** argument. The *self* variable or argument is used to access the instance variables (or object variables) of an object.

__init__() Method (Constructor)

The __init__() is a special method inside a class. It serves as the constructor. It is automatically executed when an object of the class is created. This method is used generally to initialize the instance variables, although any code can be written inside it.

Syntax of __init__() method is as follows:

```
def  __init__(self, arg1, arg2, ...):
        statement1
        statement2
        ...
        statementN
```

For our *Student* class, the __init__() method will be as follows:

```
def  __init__(self, rno, n):
        #regdno and name are instance variables
        self.regdno = rno
        self.name = n
```

__del__() Method (Destructor)

The __del__() method is a special method inside a class. It serves as the destructor. It is executed automatically when an object is going to be destroyed. We can explicitly make the __del__() method to execute by deleting an object using ***del*** keyword.

Syntax of __del__() method is as follows:

```
def  __del__(self):
        statement1
        statement2
        ...
        statementN
```

For our *Student* class, __del__() method can be used as follows:

```
def  __del__(self):
        print("Student object is being destroyed");
```

#Example Program on Constructor and Destructor

```
class Student:
        #branch is a class variable
        branch = "CSE"
        def  __init__(self, rno, n):      #This is constructor
                self.regdno = rno
                self.name = n
        def  __del__(self):               #This is destructor
                print("The object is going to be deleted")
        def  print_student_details(self):
                print("Student regd.no. is", self.regdno)
                print("Student name is", self.name)
                print("Student branch is", Student.branch)
s = Student("501", "Ramu")
s.print_student_details()
```

Output:
Student regd.no. is 501
Student name is Ramu
Student branch is CSE
The object is going to be deleted

Private Methods

Like private variables, we can also create private methods. When there is a need to hide the internal implementation of a method, we can mark the method as private. It is not recommended to access a private method outside the class. But if needed, we can access a private method outside the class as follows:

objectname._classname__privatemethod(arg1, arg2, ...)

Following example demonstrates a private method:

```
class Box:
        def  __init__(self, l, b):
                self.length = l
                self.breadth = b
        #Private method
        def  __printdetails(self):
                print("Length of box is:", self.length)
                print("Breadth of box is:", self.breadth)
        def  display(self):
                self.__printdetails()
b = Box(10, 20)
b. display() #recommended
b._Box__printdetails() #not recommended
```

Output:
Length of box is: 10
Breadth of box is: 20
Length of box is: 10
Breadth of box is: 20

Static Methods

A static method is a method marked with ***staticmethod*** decorator. Static methods are the methods that are bound to a class rather than its object. They do not require a class instance creation. So, they are not dependent on the state of the object. Static method does not have any additional arguments like ***self*** and ***cls***. Following example demonstrates a static method:

```
class Box:
        sides = None
        def  __init__(self, l, b):
                self.length = l
                self.breadth = b
        def  display(self):
                print("Length of box is:", self.length)
                print("Breadth of box is:", self.breadth)
        @staticmethod
        def  init_sides(s):
                sides = s
                print("Sides of box =", sides)
```

```
Box.init_sides(4)   #calling the static method using Class name
b = Box(10,20)
b. display()
```

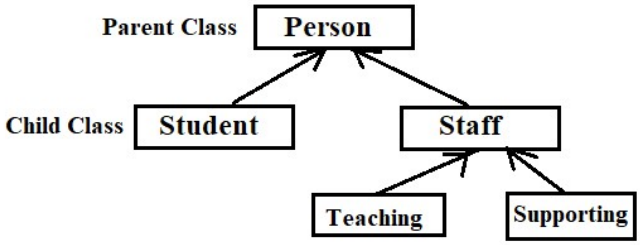
Output:
Sides of box = 4
Length of box is: 10
Breadth of box is: 20

Inheritance

Creating a new class from existing class is known as inheritance. (or)
The process of acquiring the properties of one class into another class is called as Inheritance.

Parent Class (or Base Class): The class from which features are inherited is called Base class.
Child Class (or Derived Class or Sub Class): The class into which features are derived is called Derived class.

Inheritance promotes reusability of code by reusing already existing classes. Inheritance is used to implement ***is-a*** relationship between classes.
Following hierarchy is an example representing inheritance between classes:



Syntax for creating a derived class is as follows:

```
class  DerivedClass(BaseClass):
        statement1
        statement2
        ...
        statementN
```

```
class Person:
        def  __init__(self, fname, lname):
                self.firstname = fname
                self.lastname = lname
        def  printname(self):
                print(self.firstname, self.lastname)

class Student(Person):
        pass            # In this example, we are not adding any additional members in the derived class

x = Person("John", "Doe")
x.printname()
y = Student("Mike", "Olsen")
y.printname()
```

Add the __init__() Method:
So far we have created a child class that inherits the properties and methods from its parent. We want to add the __init__() method to the child class (instead of the ***pass*** keyword). When we add the __init__() method, the child class will no longer inherit the parent's __init__() method. To use the parent's __init__() function, add a call to the parent's __init__() function.

Note: The child's __init__() function overrides the inheritance of the parent's __init__() function.

#Example Program on Inheritance

```
class Person:
        def __init__(self, fname, lname):
                self.firstname = fname
                self.lastname = lname

        def printname(self):
                 print(self.firstname, self.lastname)

class Student(Person):
        def __init__(self, fname, lname, year):
                super().__init__(fname, lname)
                 self.graduationyear = year

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

x = Person("Rama", "Raju")
x.printname()

y = Student("Rakesh", "Reddy", "2020")
y.printname()
y.welcome()
```

Polymorphism

Polymorphism means having many forms.

The word Polymorphism is a combination of two Greek words, ***Poly*** means ***many*** and ***Morphe*** means ***form***. In programming, polymorphism refers to the same object exhibiting different forms and behaviors.

In Python, ***method overriding, operator overloading*** are different ways of implementing polymorphism.

Example1:

```
# In-built polymorphic functions
# len() being used for a string
print(len("geeks"))

# len() being used for a list
print(len([10, 20, 30]))
```

Example2:

```
# User-defined polymorphic functions
def  add(x, y, z = 0):
    return x + y + z

print(add(2, 3))
print(add(2, 3, 4))
```

Method Overriding

In method overriding, a base class and the derived class will have a method with the same signature. The derived class method can provide different functionality when compared to the base class method.

When a derived class method is created and the method is invoked, the derived class method executes overriding the base class method. Following example demonstrates method overriding:

```
# Method Overriding - Example
class Person:
        def __init__(self, fname, lname):
                self.firstname = fname
                self.lastname = lname

         def print_details(self):
                print(self.firstname, self.lastname)
                print("From Base Class")

class Student(Person):
        def __init__(self, fname, lname, year):
                super().__init__(fname, lname)
                self.graduationyear = year

        def print_details(self):          # print_details() method is overridden in child/derived class
                print(self.firstname, self.lastname, self.graduationyear)
                print("From Derived Class")

x = Person("Rama", "Raju")
x.print_details()

y = Student("Rakesh", "Reddy", "2020")
y.print_details()
```

Method Overloading

- Class having multiple methods with the same name, but with different arguments.
- ***Python does not support method overloading by default***.
- The problem with method overloading in Python is that we may overload the methods but can only use the latest defined method.

```
def add(a, b):
   return a+b

def add(a, b, c):
   return a+b+c

#print(add(10,20))   #Error
print(add(10,20,30))
```

Output: 60

Operator Overloading

Operator Overloading means giving extended meaning beyond their predefined operational meaning.

Example: Operators '+', '*' performing different operations depending upon the context.

10+20 "vit" + "bvrm" [10,20] + [30,40]	10*20 [10,20]*3

How to overload the operators in Python?

When we use an operator on user defined data types, then automatically a ***special function or magic function*** associated with that operator is invoked. Operators work according to that behaviour defined in the magic functions. Changing this magic method's code, we can give extra meaning to the operator.

The magic function name for operator + is __add()__

Example: Overloading operator + to add two complex numbers

```
#Example - Operator Overloading
class complex:
        def __init__(self, a, b):
                self.a = a
                self.b = b
        # adding two objects
        def __add__(self, other):
                return self.a + other.a, self.b + other.b
Ob1 = complex(1, 2)
Ob2 = complex(3, 4)
print(Ob1 + Ob2)
```

The following tables give magic function names for few operators.

Operator	Magic Method
+	__add__(self, other)
–	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)
>>	__rshift__(self, other)
<<	__lshift__(self, other)
&	__and__(self, other)
|	__or__(self, other)
^	__xor__(self, other)

Comparison Operators

Operator	Magic Method
<	__LT__(SELF, OTHER)
>	__GT__(SELF, OTHER)
<=	__LE__(SELF, OTHER)
>=	__GE__(SELF, OTHER)
==	__EQ__(SELF, OTHER)
!=	__NE__(SELF, OTHER)

Exception Handling

Introduction

An error is an abnormal condition that results in unexpected behavior of a program. Common kinds of errors are syntax errors and logical errors. Syntax errors arise due to poor understanding of the language. Logical errors arise due to poor understanding of the problem and its solution.

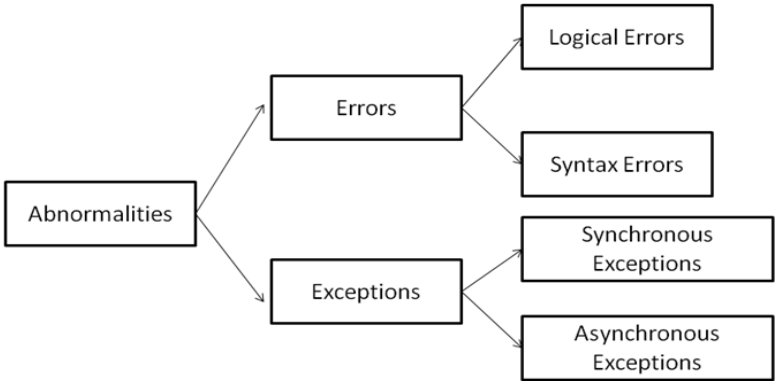
Anomalies that occur at runtime are known as exceptions.
Exceptions are of two types:

- synchronous exceptions
- asynchronous exceptions

Synchronous exceptions are caused due to mistakes in the logic of the program and can be controlled. Asynchronous exceptions are caused due to hardware failure or operating system level failures and cannot be controlled.

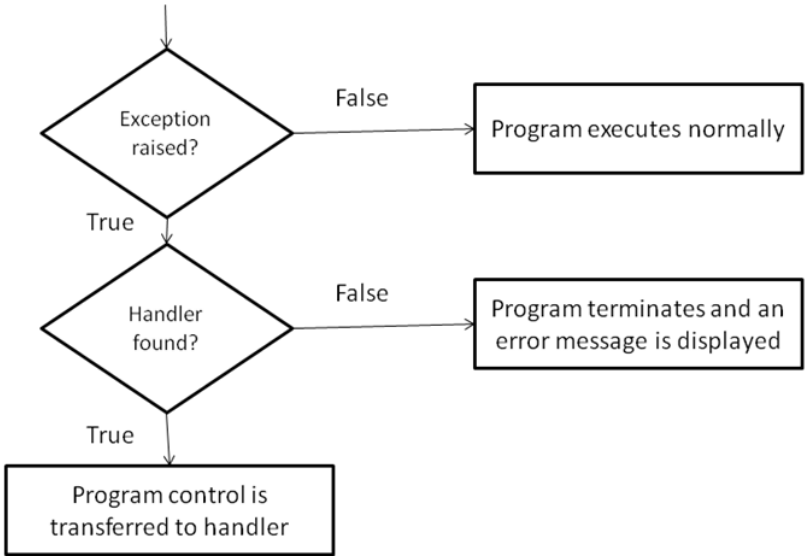
Examples of synchronous exceptions are: divide by zero, array index out of bounds, etc.
Examples of asynchronous exceptions are: out of memory error, memory overflow, memory underflow, disk failure, etc.

Overview of errors and exceptions in Python is as follows:



Handling Exceptions

Flowchart for exception handling process is as follows:



We can handle exceptions in Python code using ***try*** and ***except*** blocks. Statements which can raise exceptions are placed in ***try*** block. Code that handles the exception is placed in ***except*** block. The code that handles an exception is known as ***exception handler*** and this process is known as ***exception handling***.

try and except

Syntax for using try and except for exception handling is as follows:

```
try:
        statement(s)
except  ExceptionName:
        statement(s)
```

Following is an example for handling divide by zero exception using try and except blocks:

```
numerator = int(input())
denominator = int(input())
try:
        quotient = numerator / denominator
        print("Quotient:", quotient)
except  ZeroDivisionError:
        print("Denominator cannot be zero")
```

Input: 10 0 **Output:** Denominator cannot be zero	**Input:** 10 2 **Output:** Quotient: 5.0

Multiple Except Blocks

Often, there will be a need to handle more than one exception raised by a try block. To handle multiple exceptions, we can write multiple except blocks as shown below:

```
try:
        statement(s)
except  ExceptionName1:
        statement(s)
except  ExceptionName2:
        statement(s)
...
```

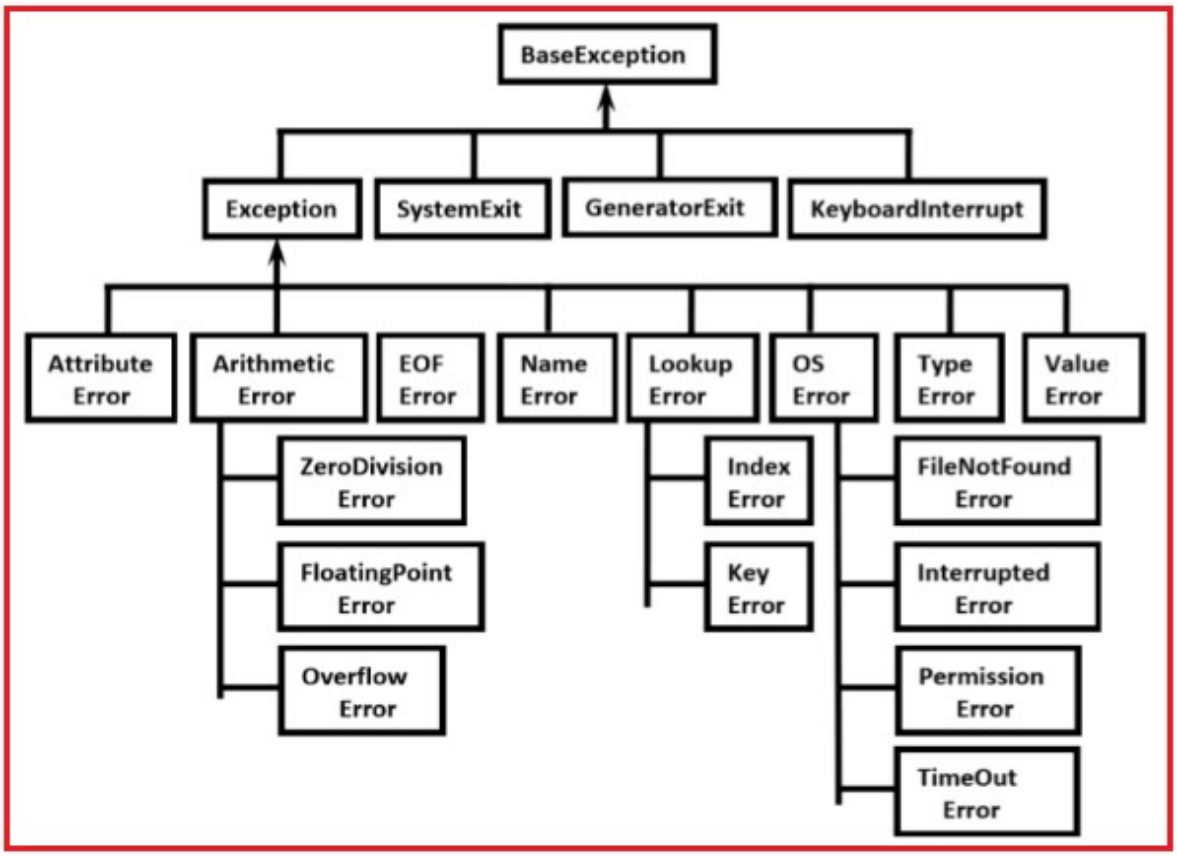
Following are the examples for handling multiple exceptions using multiple except blocks:

#Example1 *try:* *f = open('missing.txt')* *except FileNotFoundError:* *print('File not found')* *except OSError:* *print('Exception occured')* ***Output:*** **(Assumption:** ***missing.txt*** file is not available in the current working directory***)*** **File not found**	***#Example2*** *numerator = int(input())* *denominator = int(input())* *try:* *quotient = numerator / denominator* *print("Quotient:", quotient)* *except NameError:* *print("You are using a variable which is not declared")* *except ValueError:* *print("Invalid value")* *except ZeroDivisionError:* *print("Denominator cannot be zero")*

	Input: 10 0 **Output:** Denominator cannot be zero

Note: *When we are using multiple except blocks, we have to order the except blocks from more specific exceptions to generic exceptions. Otherwise (if we specify generic exception blocks at the beginning), those exceptions will match and the later part (specific exception blocks) will not be executed.*

The following figure shows the exception hierarchy.



Multiple Exceptions in a Single Block

We can handle multiple exceptions using a single except block as follows:

```
try:
        statement(s)
except(ExceptionName1, ExceptionName2,..):
        statement(s)
```

Following is an example which demonstrates handling multiple exceptions using a single except block:

```
numerator = int(input())
denominator = int(input())
try:
        quotient = numerator / denominator
        print("Quotient:", quotient)
except (NameError, ValueError, ZeroDivisionError):
        print("Some exception occured")
```

Input:
10
0
Output:
Some exception occured

Handle Any Exception

In some cases, we might want to execute the same code (handler) for any type of exception. Such common handler can be created using except as follows:

try:
    statement(s)
except:
    statement(s)
Following is an example which demonstrates handling any exception with a single except block:

```
numerator = int(input())
denominator = int(input())
try:
    quotient = numerator / denominator
    print("Quotient:", quotient)
except:
    print("Some exception occured")
```

else Clause (or Block)

The try and except blocks can be followed by an optional *else* block. ***The code in else block executes only when there is no exception in the try block***. The else block can be used to execute housekeeping code like code to free the resources that are being used.

```
#Example – else block
def divide(x, y):
   try:
      result = x // y
   except ZeroDivisionError:
      print("You are dividing by zero ")
   else:
      print("Your answer is :", result)

divide(3, 2)
divide(3, 0)
```

Output:
Your answer is : 1
You are dividing by zero

finally Clause (or Block)

A try block must be followed by one or more except blocks or one finally block. A finally block contains code that executes irrespective of whether an exception occurs or not. The finally block is generally used to write resource freeing code.
Syntax:

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

 # Example program on exception handling with else, finally blocks

```
def divide(x, y):
    try:
        result = x // y
    except ZeroDivisionError:
        print("U r dividing by zero ")
    else:
        print("Your answer is :", result)
    finally:
        print('This is always executed')

divide(3, 2)
divide(3, 0)
```

Output:

```
Your answer is : 1
This is always executed
U r dividing by zero
This is always executed
```

Handling Exceptions in Functions

We can use try and except blocks inside functions as we are using until now. In the try block, we can call a function. If this function raises an exception, it can be handled by the except block which follows the try block. Following example demonstrates handling exceptions in functions:

```
def   div(num, denom):
        return num/denom
try:
        div(30, 0)
except  ZeroDivisionError:
        print("Denominator cannot be zero")
```

Output:

```
Denominator cannot be zero
```

Raising Exceptions

We can raise exceptions manually, even though there are no exceptions, using the ***raise*** keyword. Syntax of raise statement is as follows:

```
        raise ExceptionName
```

ExceptionName is the type of exception we want to raise.
Following is an example for demonstrating ***raise*** keyword:

```
try:
        raise NameError
except  NameError:
        print("Name error")
```

In the above code ***NameError*** exception is raised by the raise statement and is handled by the except block.

User-defined Exceptions

If the pre-defined exceptions doesn't handle your custom error condition, we can create our own exceptions. Such exceptions are known as ***user-defined*** or ***custom exceptions***.

Steps in creating custom exceptions:

1. Create a new class and extend ***Exception*** class.
2. Raise the custom exception.
3. Handle the custom exception.

Following example demonstrates creating and using a user-defined exception:

```
#NegativeError is the custom exception
class  NegativeError(Exception):
        def  __init__(self, value):
                self.value = value
try:
        raise  NegativeError(-20)
except  NegativeError as ne:
        print(ne.value,"is negative.")
```

Built-in Exceptions

There are several built-in or pre-defined exceptions in Python. Python automatically recognizes the built-in exceptions and handles them appropriately. Following are some of the built-in exceptions in Python:

Exception	**Description**
Exception	Base class for all exceptions
SystemExit	Raised by sys.exit() function
ArithmeticError	Base class for errors generated by mathematical calculations
OverflowError	Raised when the maximum limit of a numeric type exceeds
FloatingPointError	Raised when a floating point error could not be raised
ZeroDivisionError	Raised when a number is divided by zero
AttributeError	Raised when attribute reference or assignment fails
EOFError	Raised when end-of-file is reached or there is no input for input() function
ImportError	Raised when an import statement fails
LookupError	Base class for all lookup errors
IndexError	Raised when an index is not found in a sequence
KeyError	Raised when a key is not found in the dictionary
NameError	Raised when an identifier is not found in local or global namespace
IOError	Raised when input or output operation fails
SyntaxError	Raised when there is syntax error in the program
ValueError	Raised when the value of an argument is invalid
TypeError	Raised when two incompatible types are used in an operation

Exception handling (summary)

try: This block will test the excepted error to occur
except: Here you can handle the error
else: If there is no exception then this block will be executed
finally: Finally block always gets executed whether exception is generated or not

Syntax:

```
try:
    # Some Code....
except:
    # optional block
    # Handling of exception (if required)
else:
    # execute if no exception
finally:
    # Some code .....(always executed)
```

Object Oriented Programming (OOP) vs. Procedural Oriented Programming (POP)

- Object Oriented Programming deals with objects and their properties.
- Major concepts/features of OOPs:
  - Class/objects
  - Abstraction
  - Encapsulation
  - Polymorphism
  - Inheritance
- Procedural Oriented Programming deals with functions. Programs are divided into functions.

Following are the important differences between OOP and POP.

Key	OOP	POP
Approach	OOP follows ***bottom up*** approach.	POP follows ***top down*** approach.
Division	A program is divided to objects and their interactions.	A program is divided into functions and their interactions.
Inheritance	Inheritance is supported.	Inheritance is not supported.
Access control	Access control is supported via access modifiers.	No access modifiers are supported.
Data Hiding	Encapsulation is used to hide data.	No data hiding present. Data is globally accessible.
Example	Python, C++, Java	C, Pascal

Features/Characteristics of Object Oriented Programming (OOP)

- **Class/objects:**
  Object-oriented programming paradigm is based upon the concept of Classes and Objects. An object is an instance of a class. A class creates a new type and provides a blueprint or a template used for creating objects.
  # Provide a suitable example for Class and Object
- **Abstraction:**
  Hiding the complexity and providing only the necessary details. It is one of the important features of OOP.
- **Encapsulation:**
  Bundling (wrapping) of data, along with the methods that operate on that data, into a single unit. Encapsulation provides data hiding
- **Polymorphism:**
  Polymorphism means having many forms. The word Polymorphism is a combination of two Greek words, ***Poly*** means ***many*** and ***Morphe*** means ***form***. In programming, polymorphism refers to the same object exhibiting different forms and behaviors. Polymorphism is provided by Method overloading, Operator overloading, Method overriding etc. ***Python doesn't support method overloading by default.***
- **Inheritance:**
  Creating a new class from existing class is known as inheritance. (or) The process of acquiring the properties of one class into another class is called as Inheritance.

  Parent Class (or Base Class): The class from which features are inherited is called Base class.

  Child Class (or Derived Class or Sub Class): The class into which features are derived is called Derived class.

  Inheritance promotes reusability of code by reusing already existing classes. Inheritance is used to implement ***is-a*** relationship between classes.

  Following hierarchy is an example representing inheritance between classes: