OOP and Exception Handling

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 using for creating objects. In Python we already know that everything is an object of some class. In Python, the base class for all classes is *object*.

Creating or Defining a Class

The syntax for 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 are together 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. A class definition creates a new namespace.

Creating Objects

Once a class is defined, we can create objects for 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 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 C and 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");
```

Other Special Methods

Method	Description
__repr__()	This method returns the string representation of an object. It works on any object, not just user class instances. It can be called as *repr(object)*.
__cmp__()	This method can be used to override the behavior of == operator for comparing objects of a class. We can also write our own comparison logic.
__len__()	This method is used to retrieve the length of an object. It can be called as *len(object)*.
__call__()	This method allows us to make a class behave like a function. Its instance can be called as *object(arg1,arg2,...)*
__hash__()	While working with sets and dictionaries in Python, the objects are stored based on hash value. This value can be generated using this method.
__lt__()	This method overrides the behavior of < operator.
__le__()	This method overrides the behavior of <= operator.
__eq__()	This method overrides the behavior of = operator.
__ne__()	This method overrides the behavior of != operator.
__gt__()	This method overrides the behavior of > operator.
__ge__()	This method overrides the behavior of >= operator.
__iter__()	This method can be used to override the behavior of iterating over objects.
__getitem__()	This method is used for indexing, i.e., for accessing a value at a specified index.
__setitem__()	This method is used to assign an item at a specified index.

Private Methods

Like private variables, we can also created private methods. When there is a need to hide the internal implementation of a class, 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, ...)

If needed, we can access private variables using similar syntax as above.

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
```

Built-in Functions

Following are some of the built-in functions to work with objects:

Function	Description
hasattr(obj, name)	This function checks whether the specified attribute belongs to the object or not.
getattr(obj, name[,default])	This function retrieves the value of the attribute on an object. If the default value is given it returns the that if no attribute is present. Otherwise, this function raises an exception.
setattr(obj, name, value)	This function is used to set the value of an attribute on a given object.
delattr(obj, name)	This function deletes an attribute on the given object.

Built-in Class Attributes

Every class in Python by default has some attributes. Those attributes can be accessed using dot (.) operator. These attributes are as follows:

Attribute	Description
__dict__	Gives the dictionary containing the namespace of a class.
__doc__	Gives the documentation string of the class if specified. Otherwise, it returns None.
__name__	Gives the name of the class
__module__	Gives the name of the module in which the class is defined
__bases__	Gives the base classes in inheritance as a tuple. Otherwise, it returns an empty tuple.

Garbage Collection

Python runs a process in the background which frees an object from memory when it is no longer needed or if it goes out of scope. This process is known as garbage collection. Python maintains a reference count for each object in the memory.

The reference count increases when we create aliases or when we assign an object a new name or place it in a list or other data structure. Reference count decreases when an object goes out of scope and it becomes zero when the object is deleted using *del* keyword.

Class Methods

A class method is a method which is marked with *classmethod* decorator and the first argument to the method is *cls*. A class method is called using the class name. Class methods are generally used as a factory method to create instances of the class with the specified arguments. Example of a class method is as follows:

```
class Box:
        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)
        @classmethod
        def  CreateInstance(cls, l, b):
                return cls(l, b)
b = Box.CreateInstance(10,20)
b. display()
```

Static Methods

A static method is a method marked with *staticmethod* decorator. Code that manipulates information or data of a class is placed in a static method. Static method does not depend on the state of the object. Static method can be called using class name. Static method does not have any additional arguments like *self* and *cls*.

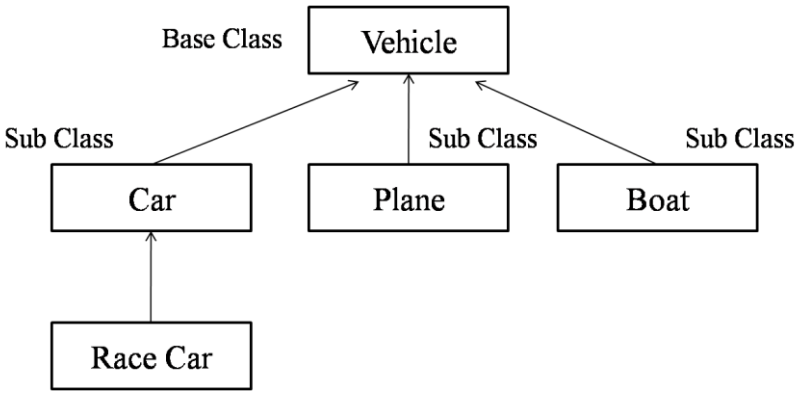
Following example demonstrates a static method:

```
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)
b = Box(10,20)
b. display()
```

Inheritance

Creating a new class from existing class is known as inheritance. The class from which features are inherited is known as base class and the class into which features are derived into 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
```

Following example demonstrates inheritance in Python:

```
class  Shape:
        def  __init__(self): print("Shape Constructor")
        def  area(): print("Area of shape")
        def  peri(): print("Perimeter of shape")
class  Circle(Shape):
        def  __init__(self, r):
                Shape.__init__(self) #Calling parent constructor
                self.radius = r
                print("Circle Constructor")
        def  area(self): print("Area of circle is:", 3.142*self.radius*self.radius)
c = Circle(5)
c.area()
```

Polymorphism

Polymorphism means having different forms. Based on the context, a variable, function, or an object can have different meaning. As inheritance is related to classes, polymorphism is related to methods. In Python, method overriding is one way of implementing polymorphism.

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:

```
class  Shape:
        def  __init__(self): print("Shape Constructor")
        def  area(self): print("Area of shape")
        def  peri(self): print("Perimeter of shape")
class  Circle(Shape):
        def  __init__(self, r):
                Shape.__init__(self) #Calling parent constructor
                self.radius = r
                print("Circle Constructor")
        def  area(self):
                print("Area of circle is:", 3.142*self.radius*self.radius)
c = Circle(5)
c.area() #Circle class area method overrides Shape class area method
```

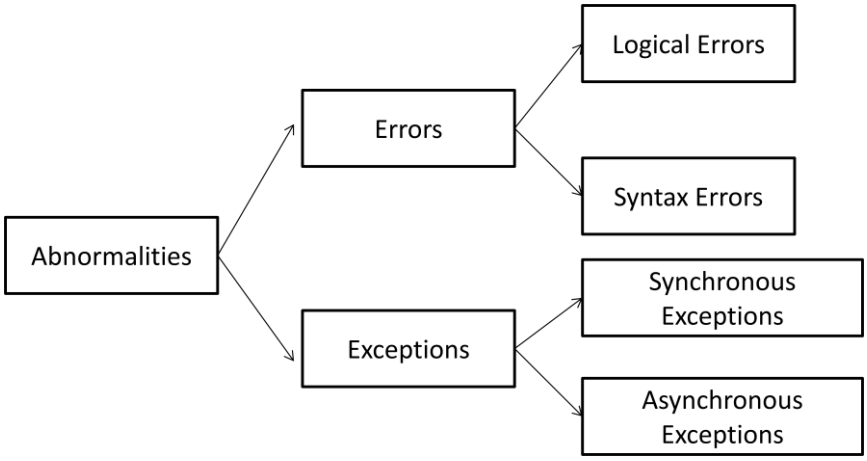
Exception Handling

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.

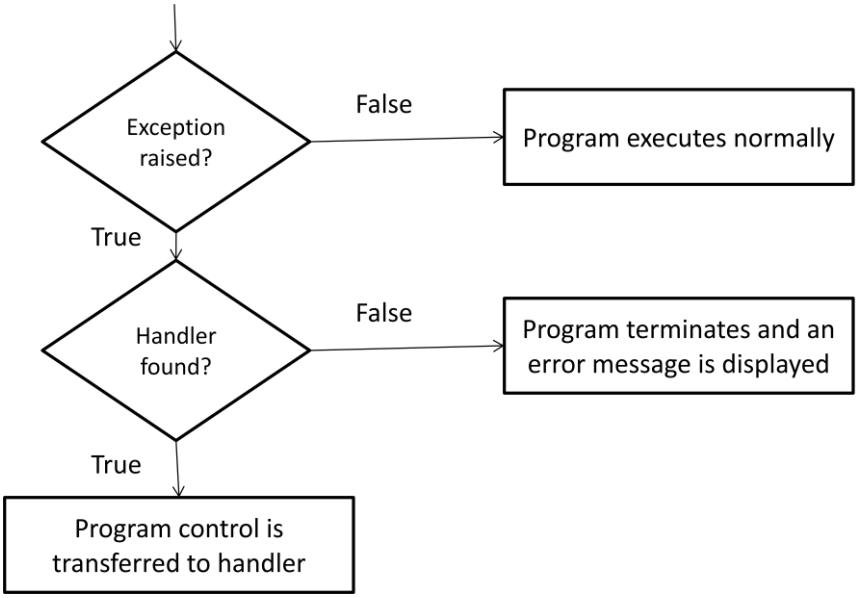
Anomalies that occur at runtime are known as exceptions. Exceptions are of two types: synchronous exceptions and asynchronous exceptions. Synchronous exceptions are caused due to mistakes in the logic of the program and can be controlled. Asynchronous exceptions are caused due to hardware failure or 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 is an example for handling multiple exceptions using multiple except blocks:

```
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")
```

In the previous example for input 10 and 0, the output displayed is “Denominator cannot be zero”. When the divide by zero exception was triggered in try block, first two except blocks were skipped as the type of exception didn’t match with either *NameError* or *ValueError*. So, third except block was executed.

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("Denominator cannot be zero")
```

For input 20 and 0, the output for the above code is:

Denominator cannot be zero

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("Denominator cannot be zero")
```

For input 8 and 0, the output for the above code is:

Denominator cannot be zero

else Clause

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.

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 [, args [, traceback]]]
```

ExceptionName is the type of exception we want to raise. Args are the exception arguments and traceback is the traceback object used for the exception. 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.

Re-raising Exceptions

In Python, we can re-raise the exceptions in the except block by writing *raise* keyword. Following code demonstrates re-raising exceptions:

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

Output is as follows:

```
Name error
Traceback (most recent call last):
  File "test1.py", line 2, in <module>
    raise NameError
NameError
```

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 for the above program is as follows:

```
Denominator cannot be zero
```

finally 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. Syntax for finally block is as follows:

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

The finally block is generally used to write resource freeing code. We cannot write a else block along with finally block.

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
StandardError	Base class for all built-in exceptions (excluding StopIteration and SystemExit)
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
AssertionError	Raised when the assert statement fails
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
RuntimeError	Raised when the generated error does not fall into any of the above categories
NotImplementedError	Raised when an abstract method that needs to be implemented is not implemented in the derived class

TypeError	Raised when two incompatible types are used in an operation

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
        def  __str__(self):
                return repr(self.value)
try:
        raise  NegativeError(-20)
except  NegativeError  as  ne:
        print(ne.value,"is negative.")
```