Exception Handling, Inheritance and Decorator

Learn about exception handling, raise custom error about inheritance also little bit about logging

Different types of exceptions in python:

In Python, there are several built-in exceptions that can be raised when an error occurs during the execution of a program. Here are some of the most common types of exceptions in Python:

- **SyntaxError:** This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.
- **TypeError**: This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.
- **NameError**: This exception is raised when a variable or function name is not found in the current scope.
- **IndexError**: This exception is raised when an index is out of range for a list, tuple, or other sequence types.
- **KeyError**: This exception is raised when a key is not found in a dictionary.
- **ValueError**: This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.
- **AttributeError**: This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.
- **IOError**: This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.
- **ZeroDivisionError**: This exception is raised when an attempt is made to divide a number by zero.
- **ImportError**: This exception is raised when an import statement fails to find or load a module.

These are just a few examples of the many types of exceptions that can occur in Python. It's important to handle exceptions properly in your code using try-except blocks or other error-handling techniques, in order to gracefully handle errors and prevent the program from crashing

```
#Syntax error
varage = 25
# check that You are eligible to
if(varage > 21)
print("You are eligible to purchase car  ")
#output

File "<string>", line 6

  if(varage > 21 )

             ^

SyntaxError: expected ':'

#Exception error
```

```
samplemark = 1005

 # perform division with 0

vara = samplemark / 0

print(“The result is:-”,vara)

#output

Traceback (most recent call last):

File "<string>", line 5, in <module>

ERROR!

ZeroDivisionError: division by zero

#Type Error

x = 5

y = "hello"

z = x + y  # Raises a TypeError: unsupported operand type(s) for +: 'int' and 'str'

Traceback (most recent call last):

ERROR!

File "<string>", line 3, in <module>

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Try and Except Statement – Catching Exceptions

Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed. For example, we can add IndexError in the above code. The general syntax for adding specific exceptions are –

```
try:
   # statement(s)
except IndexError:
   # statement(s)
except ValueError:
   # statement(s)
```

```
x = 5
y = "hello"
try:
   z = x + y
except TypeError:
   print("Error: cannot add an int and a str")
```

```
Error: cannot add an int and a str
```

```
# Program to handle multiple errors with one except statement
def fun(vara):
   if vara < 4:
      # throws ZeroDivisionError for a = 3
      varb = vara/(vara-3)
   # throws NameError if a >= 4
   print("Value of b = ", varb)
try:
   fun(3)
   fun(5)
except ZeroDivisionError:
   print("ZeroDivisionError Occurred and Handled")
except NameError:
   print("NameError Occurred and Handled")
```

```
ZeroDivisionError Occurred and Handled
```

Try with Else Clause

In Python, you can also use the else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

```
# Program to depict else clause with try-except
def mysamplefun(a , b):
   try:
      c = ((a+b) / (a-b))
   except ZeroDivisionError:
      print ("a/b not possible if b is 0")
   else:
      print (c)

# above function
mysamplefun(7, 3)
```

```
mysamplefun(8, 3)
```

```
2.5
2.2
```

```
# Aother way to write the exception code
def divide(x, y):
   try:
      # Floor Division : Gives only Fractional Part as Answer
      result = x // y
      print("Yeah ! Your answer is :", result)
   except Exception as e:
     # By this way we can know about the type of error occurring
      print("The error is: ",e)

divide(3, "GFG")
divide(3,0)
```

```
The error is:  unsupported operand type(s) for //: 'int' and 'str'
The error is:  integer division or modulo by zero
```

finally Keyword in Python

Python provides a keyword finally, which is always executed after the try and except blocks. The final block always executes after the normal termination of the try block or after the try block terminates due to some exception.

Syntax:

```
try:
   # Some Code....

except:
   # optional block
   # Handling of exception (if required)

else:
   # execute if no exception

finally:
   # Some code .....(always executed)

# Python program to demonstrate finally

# No exception Exception raised in try block
```

```
try:
    k = 5//0  # raises divide by zero exception.
    print(k)

# handles zerodivision exception
except ZeroDivisionError:
    print("Can't divide by zero")

finally:
    # this block is always executed
    # regardless of exception generation.
    print('This is always executed')
```

```
Can't divide by zero
This is always executed
```

Raising Exception

The **raise statement** allows the programmer to force a specific exception to occur. The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).

```
# Program to depict Raising Exception

try:

    raise NameError("Hi there")  # Raise Error

except NameError:

    print ("An exception")

    raise  # To determine whether the exception was raised or not

#output

An exception

ERROR!

Traceback (most recent call last):

  File "<string>", line 4, in <module>

NameError: Hi there
```

Assertions in Python

- Assertions are statements that assert or state a fact confidently in your program. For example, while writing a division function, you're confident the divisor shouldn't be zero, you assert divisor is not equal to zero.
- Assertions are simply boolean expressions that check if the conditions return true or not. If it is true, the program does nothing and moves to the next line of code. However, if it's false, the program stops and throws an error.
- It is also a debugging tool as it halts the program as soon as an error occurs and displays it.
- Python has built-in assert statement to use assertion condition in the program. assert statement has a condition or expression which is supposed to be always true. If the condition is false assert halts the program and gives an AssertionError.

Syntax for using Assert in Pyhton:

```
assert <condition>
assert <condition>,<error message>
```

```
def avg(marks):
    assert len(marks) != 0
    return sum(marks)/len(marks)

mark1 = []
print("Average of mark1:",avg(mark1))
```

When we run the above program, the output will be: AssertionError

```
def avg(marks):
    assert len(marks) != 0,"List is empty."
    return sum(marks)/len(marks)

mark2 = [55,88,78,90,79]
print("Average of mark2:",avg(mark2))

mark1 = []
print("Average of mark1:",avg(mark1))
```

When we run the above program, the output will be:

Average of mark2: 78.0

AssertionError: List is empty.

```
# define Python user-defined exceptions
class InvalidAgeException(Exception):
   "Raised when the input value is less than 18"
   pass

# you need to guess this number
number = 18

try:
   input_num = int(input("Enter a number: "))
   if input_num < number:
      raise InvalidAgeException
   else:
      print("Eligible to Vote")

except InvalidAgeException:
   print("Exception occurred: Invalid Age")
```

Output

If the user input input_num is greater than **18**,
Enter a number: 45
Eligible to Vote
If the user input input_num is smaller than **18**,
Enter a number: 14
Exception occurred: Invalid Age
In the above example, we have defined the custom exception InvalidAgeException by creating a new class that is derived from the built-in Exception class.
Here, when input_num is smaller than **18**, this code generates an exception.
When an exception occurs, the rest of the code inside the try block is skipped.
The except block catches the user-defined InvalidAgeException exception and statements inside the except block are executed.

Constructors

Constructors are generally used for instantiating an object. The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created. In Python the __init__() method is called the constructor and is always called when an object is created.
Syntax

```
def __init__(self):
   # body of the constructor
```

Example1

```
class GeeksTakesWorld:
    # default constructor
   def __init__(self):
      self.savy = "GeekorNerds"
    # a method for printing data members
   def print_gse(self):
      print(self.savy)

sampleobj = GeekTakesWorld()
sampleobj.print_gse()
```

Example2

```
class Addition:
   var_firstno = 0
   var_secondno = 0
   var_result = 0
    # parameterized constructor
   def __init__(self, f, s):
      self.first = f
      self.second = s

   def funcshow(self):
      print("First number = " + str(self.first))
      print("Second number = " + str(self.second))
      print("Addition of two numbers = " + str(self.answer))

   def functotal(self):
      self.answer = self.first + self.second


# creating object of the class this will invoke parameterized constructor
sampledr = Addition(1000, 2000)
# perform Addition on
sampledr.functotal()
# display result of
sampledr.funcshow()
```

Python has a garbage collector that handles memory management automatically so need for destructor.The __del__() method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

Syntax

```
def __del__(self):
  # body of destructor
```

Example

```
class A:
   def __init__(self, bb):
      self.b = bb

class B:
   def __init__(self):
      self.a = A(self)
   def __del__(self):
      print("die")

def samplefunc():
   b = B()

samplefun()
#output die
```

Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.
Parent class is the class being inherited from, also called base class.
Child class is the class that inherits from another class, also called derived class. Any class can be a parent class, so the syntax is the same as creating any other class:
Types of Inheritance in Python

- Single Inheritance – It enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code
- Multiple Inheritance – When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class
- Multilevel Inheritance – In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and a grandfather.
- Hierarchical Inheritance – When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. In this we have a parent (base) class and two child classes.

Some Example

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

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

class Student(Person):
```

```
  pass

samplobjx = Student("Ravi", "Kisan")
samplobjx.showname()
# Ravi Kisan

# Error Example
class A:
   def __init__(self, n='Rahul'):
      self.name = n

class B(A):
   def __init__(self, roll):
      self.roll = roll

object = B(23)
print(object.name)

#Output
Traceback (most recent call last):
  File "./prog.py", line 10, in <module>
AttributeError: 'B' object has no attribute 'name'
```

Using Super Function

The super function is a built-in function that returns the objects that represent the parent class. It allows to access the parent class's methods and attributes in the child class.

```
Example

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

  def parentdisplay(self):
    print(self.name, self.age)

# child class
class Student(Person):
  def __init__(self, name, age):
    self.sName = name
    self.sAge = age
    # inheriting the properties of parent class
```

```
    super().__init__("Rahul", age)

  def displayInfo(self):
    print(self.sName, self.sAge)

sampleobj = Student("Mayank", 23)
sampleobj.parentdisplay()
sampleobj.displayInfo()
#Output
```
Rahul 23
Mayank 23

Example2

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

  def Maindispl(self):
    print(self.name, self.age)

# child class
class Student(Person):
  def __init__(self, name, age, dob):
    self.sName = name
    self.sAge = age
    self.dob = dob
    # inheriting the properties of parent class
    super().__init__("Rahul", age)

  def Infodispl(self):
    print(self.sName, self.sAge, self.dob)

sampleobj = Student("Mayank", 23, "16-03-2000")
sampleobj.Maindispl()
sampleobj.Infodispl()
#output
```
Rahul 23
Mayank 23 16-03-2000

```
# Python example to show the working of multiple inheritance

class Base1(object):
```

```
    def __init__(self):
        self.str1 = "Geek1"
        print("Base1")


class Base2(object):
    def __init__(self):
        self.str2 = "Geek2"
        print("Base2")


class Derived(Base1, Base2):
    def __init__(self):

        # Calling constructors of Base1
        # and Base2 classes
        Base1.__init__(self)
        Base2.__init__(self)
        print("Derived")

    def printStrs(self):
        print(self.str1, self.str2)


ob = Derived()
ob.printStrs()
```

#ouptut

```
Base1
Base2
Derived
Geek1 Geek2
```

```
# A Python program to demonstrate inheritance
# Base or Super class. Note object in bracket.

class Base(object):

    # Constructor
    def __init__(self, name):
        self.name = name

    # To get name
    def getName(self):
        return self.name
```

```
# Inherited or Sub class (Note Person in bracket)
class Child(Base):

    # Constructor
    def __init__(self, name, age):
        Base.__init__(self, name)
        self.age = age

    # To get name
    def getAge(self):
        return self.age

# Inherited or Sub class (Note Person in bracket)


class GrandChild(Child):

    # Constructor
    def __init__(self, name, age, address):
        Child.__init__(self, name, age)
        self.address = address

    # To get address
    def getAddress(self):
        return self.address


# Driver code
g = GrandChild("Geek1", 23, "Noida")
print(g.getName(), g.getAge(), g.getAddress())

#output
Geek1 23 Noida
```

Private members of the parent class We don't always want the instance variables of the parent class to be inherited by the child class i.e. we can make some of the instance variables of the parent class private, which won't be available to the child class. In Python inheritance, we can make an instance variable private by adding double underscores before its name.

Example

```
class C(object):
    def __init__(self):
        self.c = 21
```

```
        # d is private instance variable
        self.__d = 42

class D(C):
    def __init__(self):
        self.e = 84
        C.__init__(self)

object1 = D()

# produces an error as d is private instance variable
print(object1.c)
print(object1.__d)

#Output
AttributeError: type object 'D' has no attribute 'd'
```

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.

Protected members

Those members of the class that cannot be accessed outside the class but can be accessed from within the class and its subclasses. To accomplish this in Python, just follow the convention by prefixing the name of the member by a single underscore "_".

Example

```
class Base:
    def __init__(self):
        # Protected member
        self._a = 2

# Creating a derived class
```

```
class Derived(Base):
    def __init__(self):
        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling protected member of base class: ",
              self._a)
        # Modify the protected variable:
        self._a = 3
        print("Calling modified protected member outside class: ",
              self._a)


obj1 = Derived()
obj2 = Base()
# Calling protected member
# Can be accessed but should not be done due to convention
print("Accessing protected member of obj1: ", obj1._a)
# Accessing the protected variable outside
print("Accessing protected member of obj2: ", obj2._a)
#output
Calling protected member of base class:  2
Calling modified protected member outside class:  3
Accessing protected member of obj1:  3
Accessing protected member of obj2:  2
```

Private members are similar to protected members, the difference is that the class members declared private should neither be accessed outside the class nor by any base class. In Python, there is no existence of Private instance variables that cannot be accessed except inside a class. However, to define a private member prefix the member name with double underscore "__".

Python's private and protected members can be accessed outside the class through python name mangling.

```
class Base:
    def __init__(self):
        self.a = "ServiceVan"
        self.__c = "TravelMart"
# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
```

```
        Base.__init__(self)
        print("Calling private member of base class: ")
        print(self.__c)
# Driver code
obj1 = Base()
print(obj1.a)
print(obj1.c)
#Output
Traceback (most recent call last):
  File "./prog.py", line 17, in <module>
AttributeError: 'Base' object has no attribute 'c'
```

Basic Logging

Python has a built-in library, logging, for this purpose. It is simple to create a “logger” to log messages or information that you would like to see.

The logging system in Python operates under a hierarchical namespace and different levels of severity. The Python script can create a logger under a namespace, and every time a message is logged, the script must specify its severity. The logged message can go to different places depending on the handler we set up for the namespace. The most common handler is to simply print on the screen, like the ubiquitous print() function. When we start the program, we may register a new handler and set up the level of severity to which the handler will react.

There are 5 different logging levels that indicate the severity of the logs, shown in increasing severity:

DEBUG

INFO

WARNING

ERROR

CRITICAL

```
import logging
logging.basicConfig(filename = 'application_log.txt',level=logging.DEBUG,
format=' %(asctime)s - %(levelname)s - %(message)s')
```

```
logging.debug("Data Inserted Successfully")
logging.debug('Connection Closed Successfully')
file = open("./application_log.txt","r")
for record in file.readlines():
    print(record)
```

About decorator

A Python decorator is a function that takes in a function and returns it by adding some functionality.
In fact, any object which implements the special __call__() method is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.
Basically, a decorator takes in a function, adds some functionality and returns it.

```
def make_face(func):
  def inner():
    print("In progress")
    func()
  return inner


def make_smiley():
  print("Can’t make missing module")
# Output: Can’t make missing module
```

Here, we have created two functions:
Make_smiley() that prints "Can’t make missing module"
make_face() that takes a function as its argument and has a nested function named inner(), and returns the inner function.We are calling the make_smiley() function normally, so we get the output " Can’t make missing module ". Now, let's call it using the decorator function.

```
def make_face(func):
  # define the inner function
  def inner():
    # add some additional behavior to decorated function
    print("In progress")
    # call original function
    func()
  # return the inner function
  return inner
# define make smiley function
def make_smiley():
  print("Module is found in progress")
```

```
# decorate the ordinary function
decorated_func = make_face(make_ smiley)
# call the decorated function
decorated_func()
#output
In progress
Module is found in progress
```

Instead of assigning the function call to a variable, Python provides a much more elegant way to achieve this functionality using the @ symbol. For example

```
def sample_inject(func):
    def inner():
        print("I got injected")
        func()
    return inner

@sample_inject
def volaman():
    print("mango man")

volaman()
#output
I got injected
mango man

def smart_divide(func):
    def inner(a, b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("Whoops! cannot divide")
            return

        return func(a, b)
    return inner

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

divide(2,5)
divide(2,0)
#Output
I am going to divide 2 and 5
0.4
I am going to divide 2 and 0
Whoops! cannot divide
```