Error Handling & Operator Overloading in Python

Lesson Plan

PW SKILLS

Exception Handling is a crucial concept in python which allows us to gracefully handle and manage the errors and exceptional situations which may occur during the program execution.

What is an Exception?

Exceptions is a term used for any abnormal or unexpected event happening. In python also, an exception is an abnormal or unexpected occurrence of an event occurred during program execution. Exceptions are a bad thing for a program as they disrupts the normal flow of the code and results in errors.

There are various reason an exception may occur during program execution, these are generally semantical or logical errors.

For example, an exception may occur when you are trying to open a file in read mode, when the file does not even exist. Some examples are:

1. invalid input
2. file not found
3. division by zero
4. attempting to access a non-existent variable.

Different types of exceptions in python:

In Python, there are several built-in Python 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.

Following cells of code presents the example of the exception handling in python.

```
In [ ]: # file not found error without exception
        f = open('text.txt', 'r')
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-1-97ebe980cb92> in <cell line: 1>()
----> 1 f = open('text.txt', 'r')

FileNotFoundError: [Errno 2] No such file or directory: 'text.txt'
```

This code will return an error and will stop the execution of your entire program. This is not acceptable in a production grade code and can lead to disastorous consequences with your application if sent to production.

A smart way to handle such risks is to use the concept of exception handling, by writing the code in a try and except block.

What Happens inside the 'Try and Except' Block?

In the try and except block, the code is executed in a certain manner, so that if the error is occured, the program execution must not stop, but allows us to log the error occured at any moment during the file execution.

The 'try' block tries to execute the code written inside it. If any error is encountered during the execution of this code, the execution jumps to the 'except' block and executes the code written inside this block.

Both the 'try' and except block can have the logic written under them using the conditional statements and loops like 'for' and 'while'. We can also write a simple 'print' statement inside the except block to print the encountered error while executing the 'try' block.

A more formal definition of the try-except block in 3 points can be given as follows:

1. The primary mechanism for handling exceptions in python is the 'try-except' block.
2. We enclose the code that might raise an exception inside a 'try' block.
3. If an exception occurs, python immediately jumps to the 'except' block, where we can handle the exception gracefully.

The strict syntax of the try-except block can be given as:

python Copy code

```
try:
    # Code that might raise an exception
except ExceptionType as e:
    # Code to handle the exception
```

In the example below, we have written the code to read the text file which does not exist, but not like we always write. We have written it under the try and except block this time. This does not throw us an error and stops the execution of the porgram, but completes the execution and gives us the message we mentioned in the 'except' block along with the error type.

This way of handling the error arised due to some sematical errors in our program allows us to protect the flow of the program while getting the updates about any issues found during the execution.

In real life, this practice prevents your application or program from crashing due to some error.

In [ ]:
```
try:
  f = open('test.txt', 'r')
except Exception as e:
  print("There is some issue with my code ", e)
```

```
There is some issue with my code  [Errno 2] No such file or directory: 'test.txt'
```

Assertion Error

Assertion is a programming concept used while writing a code where the user declares a condition to be true using assert statement prior to running the module. If the condition is True, the control simply moves to the next line of code. In case if it is False the program stops running and returns AssertionError Exception.
The function of assert statement is the same irrespective of the language in which it is implemented, it is a language-independent concept, only the syntax varies with the programming language.

Syntax of assertion:
assert condition, error_message(optional)
Example 1: Assertion error with error_message.

Python3

```
# AssertionError with error_message.

x = 1

y = 0

assert y != 0, "Invalid Operation" # denominator can't be
0
print(x / y)
```

Output :

Traceback (most recent call last):

```
  File "/home/bafc2f900d9791144fbf59f477cd4059.py", line 4, in
    assert y!=0, "Invalid Operation" # denominator can't be 0
AssertionError: Invalid Operation
```

The default exception handler in python will print the error_message
written by the programmer, or else will just handle the error without any message.
Both of the ways are valid.

Handling AssertionError exception:

AssertionError is inherited from Exception class, when this exception occurs and raises AssertionError there are two ways to handle, either the user handles it or the default exception handler.

In Example 1 we have seen how the default exception handler does the work.

Now let's dig into handling it manually.

Example 2

Python3

```
# Handling it manually


try:


    x = 1


    y = 0


    assert y != 0, "Invalid Operation"


    print(x / y)

# the errror_message provided by the user gets printed

except AssertionError as msg:


    print(msg)
```

Output :

Invalid Operation

Why do we need the Exception Handling

As discussed earlier in the article, if we are not using exception handling in our programs, then at the time when an exception occurs, the program would terminate absupty, displaying an error message in the console, which is not a user-friendly practice.

Exception handling allows us to handle the errors gracefully, preventing the abrupt termination of the program while recording the errors or issues encountered during the execution.

Python has a built-in hierarchy of exception, with a base class as 'BaseException', and all the exceptions are derived from this base class. We can access these exceotions through the class 'Exception' in our code, as also used in the code already mentioned in the article above.

Common exception classes include 'Exception', 'TypeError', 'ValueError', 'FileNotFound', and 'ZeroDivisioError'.
An example showing that exception handling does not stop the execution of the code abruptly can be seen through the example below.

```
In [8]: try:
          f = open("test.txt", 'r')
        except Exception as e:
          print("There is some issue with your code ", e)
          print('\n')

        print("We are continuing the execution of the Python code \n")

        a = 10
        b = 5

        print('The sum of a and b is:', a+b)
```

```
There is some issue with your code  [Errno 2] No such file or directory: 'test.txt'


We are continuing the execution of the Python code

The sum of a and b is: 15
```

Other parts of the Exception Handling code in Python

The exception handling in python does not only have the 'try' and 'except' block, but also the 'else' and 'finally' block. This section of the article will discuss these two additional blocks of the exception handling code in python.

'else' Block
The 'else' block in the code block we write for exception handling is used after the try-except block always. This block is executed if there is no exception found in the try-except block. Otherwise, if an exception is found, the except block is not executed.

An example for this is shown in the cells below.

```
In [1]: # opening a new text file in read mode, this should throw us a 'FileNotFound' error
try:
  f = open("test.txt", 'r')
except Exception as e:
  print("There is some issue with your code ", e)
  print('\n')
else:
  print("No Exception is found. The code is running fine!")
```

```
There is some issue with your code  [Errno 2] No such file or directory: 'test.txt'
```

```
In [2]: # opening a new text file in write mode. This is an executable code
try:
  f = open("test.txt", 'w')
except Exception as e:
  print("There is some issue with your code ", e)
  print('\n')
else:
  print("No Exception is found. The code is running fine!")
```

```
No Exception is found. The code is running fine!
```

As seen in the two different blocks of code above, the 'else' part is not executed when an exception is found. On the other hand, in the example where no exception is found, the 'else' block is executed as expected.

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)

Example:
The code attempts to perform integer division by zero, resulting in a ZeroDivisionError. It catches the exception and prints "Can't divide by zero." Regardless of the exception, the finally block is executed and prints "This is always executed."

```
try:

    k = 5//0

    print(k)

except ZeroDivisionError:

    print("Can't divide by zero")

Finally:

    print('This is always executed')
```

Output :

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).

This code intentionally raises a NameError with the message “Hi there” using the raise statement within a try block. Then, it catches the NameError exception, prints “An exception,” and re-raises the same exception using raise. This demonstrates how exceptions can be raised and handled in Python, allowing for custom error messages and further exception propagation.

Python3

```
try:

    raise NameError("Hi there")

except NameError:

    print ("An exception")

    raise
```

The output of the above code will simply line printed as “An exception” but a Runtime error will also occur in the last due to the raise statement in the last line. So, the output on your command line will look like

Traceback (most recent call last):

File "/home/d6ec14ca595b97bff8d8034bbf212a9f.py", line 5, in <module>
raise NameError("Hi there") # Raise Error
NameError: Hi there

Custom Exception Handling in Python

When working on a project, we might come across a situation where the code we are writing does not really returns an error through python, but some cases are an exception for us as developers. An example of such situation can be explained as follows:

For example, imagine that your application asks for the age of the user as an input, and the input value type is an integer. Now, the user may add the negative number or just zero, or a number much higher than 100 or 200. As per the system these are not errors, and thus the system will not be throwing an error to us. However, this situation is an unexpected or unwanted situation for us as developer. But, this cannot be handled with the traditional exception handling. So, how can we handle such situations through exception handling?

Custom Exception Handling is the solution for such situation.

```
In [7]: # example for situation for custom exception handling
        age = int(input('Enter Your Age: '))
Enter Your Age: -253
```

The way to create a custom exception in python is through creating a class which inherits the custom class.

Inside this class, we define the init function which has the message we want to show up whenever this exception occurs.

Now, to use this custom exception, we can use the 'raise' keyword and call the custom exception. We can also use this custom exception is the try-except block.

In [8]:
```
class validateage(Exception):
  def __init__(self, msg):
    self.msg = msg
```

In [10]:
```
def age_validate(age):
  if age<0:
    raise validateage("Age should not be less than zero")
  elif age>200:
    raise validateage("Age is too high")
  else:
    print("Age is valid")
```

In [11]:
```
try:
  age = int(input("Enter You Age "))
  age_validate(age)
except validateage as e:
  print(e)
```

```
Enter You Age -56
Age should not be less than zero
```

Class	Description
Exception	The base class for most error types
AttributeError	Raised when invalid attribute reference is made
AssertionError	Raised upon failure of an assert statement
EOFError	Raised when the end of file is reached for console or file input
ImportError	Raised when an imported module can not be found
IOError	Raided upon failure of I/O operation (i.e. opening a file)
IndexError	Raised when the index of a sequence is out of range
KeyError	Raised when a key does not exist in a dictionary
KeyboardInterrupt	Raised when the user hits interupt key (Ctrl + C)
NameError	Raised when a nonexistent identifier is used
StopIteration	Raised when nex(iterator) has no next element
TypeError	Raised when a function's parameter is given the wrong type
ValueError	Raised when a parameter has an invalid value (i.e. sqrt(-10))
ZeroDivisionError	Raised when any division by 0 is encountered

Operator Overloading

Operator overloading is a powerful feature in Python that allows developers to define custom behavior for operators such as +, -, *, and / when applied to objects of custom classes. This feature allows developers to create more intuitive and readable code, and to make their classes behave in a similar way to built-in types.

To overload an operator in Python, you need to define special methods in your class that correspond to the operator you want to overload. For example, to overload the + operator for a class called MyNumber, you would define a method called add. Similarly, to overload the — operator, you would define a method called sub, and so on. These methods are known as magic methods or dunder methods (short for "double underscore").
Here is an example of how to overload the + operator for a class called MyNumber:

```
class MyNumber:

    def __init__(self, value):

        self.value = value
def __add__(self, other):

    return MyNumber(self.value + other.value)
a = MyNumber(1)

b = MyNumber(2)

c = a + b

print(c.value) # prints 3
```

In this example, we defined a class called MyNumber, which has a single attribute called “value”. We then defined a method called add, which is invoked when the + operator is used with objects of the MyNumber class.

In this method, we add the “value” attributes of the two objects and return a new MyNumber object with the result.

It’s important to note that operator overloading only works for certain operators, and not all operators can be overloaded. You can overload most of the arithmetic operators, comparison operators, bitwise operators, and the __str__ and __repr__ methods for string representation.

Operator overloading can make your code more intuitive and readable, and can help you to create classes that behave in a similar way to built-in types. However, it’s important to use it with caution, as overloading operators can also make your code more complex and harder to understand.

In conclusion, operator overloading is a powerful feature in Python that allows developers to define custom behavior for operators when applied to objects of custom classes. It can help to create more intuitive and readable code, but it should be used with caution to avoid making the code more complex.

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

some methods for operator overloading