16. Exception Handling

October 21, 2022

1 Introduction

We can make certain mistakes while writing a program that lead to errors when we try to run it. A python program terminates as soon as it encounters an unhandled error. These errors can be broadly classified into two classes:

- Syntax errors
- Logical errors (Exceptions)

2 Syntax Errors

Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.

[1]:
```
# For Example
a = 2
if a < 3
    print(a)
```

```
  Cell In [1], line 3
    if a < 3
           ^
SyntaxError: invalid syntax
```

- As shown in the example above, an arrow indicates where the parser ran into the syntax error.

3 Exceptions

- Errors that occur at runtime (after passing the syntax test) are called exceptions or logical errors.
- For example, they occur when we
  - try to open a file(for reading) that does not exist (`FileNotFoundError`)
  - try access list element with index value greater than its length (`IndexError`)
  - try to import a module that does not exist (`ImportError`).

- Whenever these types of runtime errors occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

[2]:
```
file = open("fname.txt")
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In [2], line 1
----> 1 file = open("fname.txt")

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

Some of the common built-in exceptions in Python programming are:

Exception	Cause
`AttributeError`	Raised when attribute assignment or reference fails.
`FloatingPointError`	Raised when a floating point operation fails.
`ImportError`	Raised when the imported module is not found.
`IndexError`	Raised when the index of a sequence is out of range.
`KeyError`	Raised when a key is not found in a dictionary.
`KeyboardInterrupt`	Raised when the user hits the interrupt key (Ctrl+C).
`MemoryError`	Raised when an operation runs out of memory.
`NameError`	Raised when a variable is not found in local or global scope.
`OSError`	Raised when system operation causes system related error.
`RuntimeError`	Raised when an error does not fall under any other category.
`IndentationError`	Raised when there is incorrect indentation.
`TabError`	Raised when indentation consists of inconsistent tabs and spaces.
`SystemError`	Raised when interpreter detects internal error.
`TypeError`	Raised when a function or operation is applied to an object of incorrect
`ValueError`	Raised when a function gets an argument of correct type but improper
`ZeroDivisionError`	Raised when the second operand of division or modulo operation is

4 Exception Handling

- When exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled. If not handled, the program will crash.
- For example, let us consider a program where we have a function A that calls function B, which in turn calls function C. If an exception occurs in function C but is not handled in C, the exception passes to B and then to A.
- If never handled, an error message is displayed and program comes to a sudden unexpected halt.
- To avoid this we use exception handling

4.1 `try...except`

- In Python, exceptions can be handled using a `try` statement.
- The critical operation which can raise an exception is placed inside the try clause.
- The code that handles the exceptions is written in the `except` clause.
- We can thus choose what operations to perform once we have caught the exception.

[3]:
```
colors = ["red","green","blue"]


print(colors[4])

colors_cap = [i.upper() for i in colors]
print(colors_cap)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In [3], line 4
      1 colors = ["red","green","blue"]
----> 4 print(colors[4])
      6 colors_cap = [i.upper() for i in colors]
      7 print(colors_cap)

IndexError: list index out of range
```

[4]:
```
# For example:

colors = ["red","green","blue"]

try:
    print(colors[4])
except:
    print("You are accessing List index which does not exist!")

colors_cap = [i.upper() for i in colors]
print(colors_cap)
```

```
You are accessing List index which does not exist!
['RED', 'GREEN', 'BLUE']
```

4.2 except specific exception

[6]:
```
# For example:

colors = ["red","green","blue"]

try:
    print(colors[4])
    r = open("My_file.txt")
except IndexError:
    print("List index is out of range")


colors_cap = [i.upper() for i in colors]
print(colors_cap)
```

```
List index is out of range
['RED', 'GREEN', 'BLUE']
```

4.3 Multiple except statements

[13]:
```
# For example:

colors = ["red","green","blue"]

try:
    print(colors[4])
    r = open("My_file.txt")
except IndexError as e:
    print(e)
except:
    print("other error occured")

colors_cap = [i.upper() for i in colors]
print(colors_cap)
```

```
list index out of range
['RED', 'GREEN', 'BLUE']
```

4.4 Raising Exceptions

- Exceptions are raised when errors occur at runtime. We can also manually raise exceptions using the `raise` keyword.
- We can optionally pass values to the exception to clarify why that exception was raised.

[9]:
```
raise KeyboardInterrupt
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In [9], line 1
----> 1 raise KeyboardInterrupt

KeyboardInterrupt:
```

[10]:
```
raise KeyboardInterrupt("I did it for fun. Ha ha :D")
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In [10], line 1
----> 1 raise KeyboardInterrupt("I did it for fun. Ha ha :D")

KeyboardInterrupt: I did it for fun. Ha ha :D
```

[14]:
```
try:
    a = int(input("Enter a positive integer: "))
    if a <= 0:
        raise ValueError("That is not a positive number!")
except ValueError as ve:
    print(ve)
```

```
Enter a positive integer:  9
```

4.5 assert

- The `assert` keyword is used when debugging code.
- It lets you test if a condition in your code returns `True`, if not, the program will raise an `AssertionError`.
- You can write a message to be written if the code returns `False`

[17]:
```
num = 5
```

[18]:
```
assert num % 2 == 0
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In [18], line 1
----> 1 assert num % 2 == 0

AssertionError:
```

[19]:
```
assert num % 2 == 0, "Error occured here"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In [19], line 1
----> 1 assert num % 2 == 0, "Error occured here"

AssertionError: Error occured here
```

4.6 try with else clause

- In some situations, you might want to run a certain block of code if the code block inside try ran without any errors.
- For these cases, you can use the optional `else` keyword with the `try` statement.
- Exceptions in the `else` clause are not handled by the preceding `except` clauses.

[26]:
```
# program to print the reciprocal of even numbers

try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
    # f = open('aaaa')
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print("Reciprocal is:", reciprocal)
```

```
Enter a number:  0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In [26], line 12
     10     print("something happend")
     11 else:
---> 12     reciprocal = 1/num
     13     print("Reciprocal is:", reciprocal)

ZeroDivisionError: division by zero
```

[27]:
```
# ZeroDivisionError in above Nested try except example
# program to print the reciprocal of even numbers

try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
```

```
except:
    print("Not an even number!")
else:
    try:
        reciprocal = 1/num
        print("Reciprocal is:", reciprocal)
    except:
        pass
```

```
Enter a number:  0
```

4.7 `try...finally`

- The `try` statement can have an optional `finally` clause.
- This clause is executed no matter what, and is generally used to release external resources.

[29]:
```
try:
    f = open("test.txt","w")
    f.write("25")
except:
    print("Exception occured")
finally:
    f.close()
    print("closing file")

# This type of construct makes sure that the file is closed even if
# an exception occurs during the program execution.
```

```
closing file
```

5 User Defined Exceptions

- In Python, users can define custom exceptions by creating a new class.
- This exception class has to be derived, either directly or indirectly, from the built-in `Exception` class.
- Most of the built-in exceptions are also derived from this class.

[30]:
```
class MyError(Exception):

    # Constructor or Initializer
    def __init__(self, value):
        self.value = value

    # __str__ is to print() the value
    def __str__(self):
        return(self.value)
```

[31]:
```
try:
    raise(MyError(3*2))
except MyError as error:
    print('A New Exception occured: ',error.value)
```

```
A New Exception occured:  6
```