An Introduction to Exceptions and Exception Handling in Python

helloml.org/an-introduction-to-exceptions-and-exception-handling-in-python/

By Ajit Singh

September 24, 2021

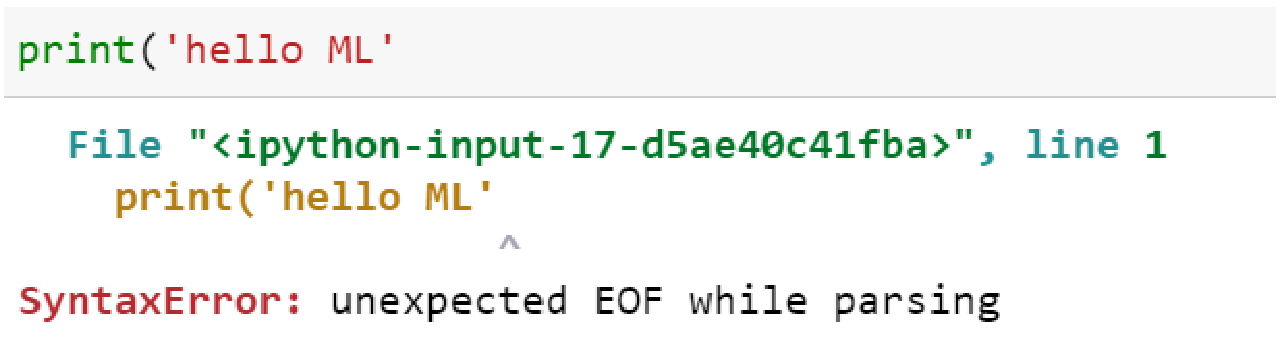
Programming gives us the capability to automate manual tasks and build useful end products. However, experience at the user-end can be ruined if the program fails or abruptly terminates during execution due to errors. As a programmer, our role is to ensure that an abstraction layer exists between our code and the end-user.

Error messages and warnings arising from our code, popping up to the user while they are running our programs are the last things we want. Not only does this ruin their experience but also makes our code vulnerable and exposed to malicious entities. Hence today we will learn all about special objects in Python called exceptions, which are used to manage errors that arise during a program's execution i.e. during run time.

Exception handling makes our programs more robust when they encounter bad data. So, unlike this banner, our program doesn't terminate abruptly, and instead, we are able to do something about the situation at hand.

Syntax Error vs Exception

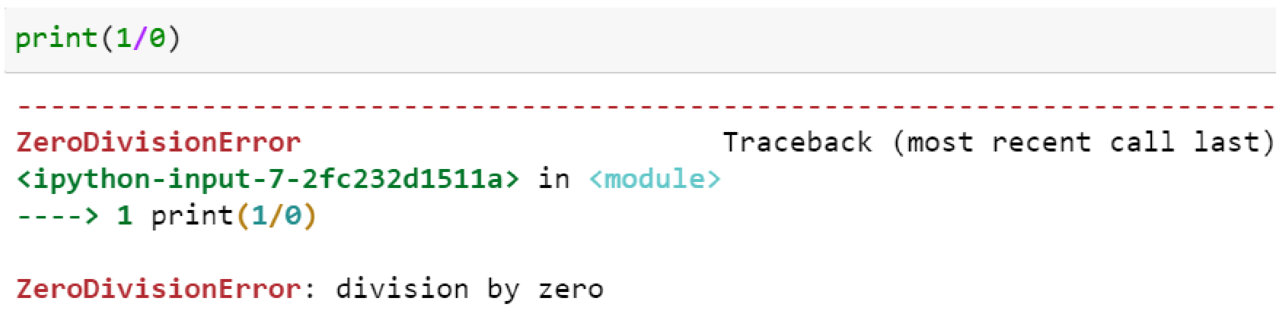
A standard form of error in Python is the **syntax error**, which is raised when the programmer violates the syntax (rules which provide the framework for writing code in a programming language) of the code. In contrast, **exceptions** are raised when the program is syntactically correct but some unexpected internal event occurs in the program which disrupts the normal expected functioning of the program (For eg, when we try to divide a number by zero or open a try to open a file that does not exist). Syntax error leads to termination of the program straight away whereas exceptions can be managed by the programmer.



```
print('hello ML'
```

```
  File "<ipython-input-17-d5ae40c41fba>", line 1
    print('hello ML'
                    ^
SyntaxError: unexpected EOF while parsing
```

A common form of syntax error: An incomplete code block.



```
print(1/0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-7-2fc232d1511a> in <module>
----> 1 print(1/0)

ZeroDivisionError: division by zero
```

A common exception: **ZeroDivisionError**, which is raised when you try to divide a number by zero.

Exception Handling in Python – Using try-except Blocks

Whenever we think that a certain piece of code may cause some unwanted problems or that an error could occur, we can wrap that section of code in a **try-except** block, which would then handle the exception that might be raised. Try-except basically tells Python to try running the code indented within the try statement and if an error is raised, run the code in the except block. Thus, the try block is used to detect the exception and the except block is used to handle it appropriately as determined by the programmer.

Here's an example of handling a situation in which an exception might occur (In this case, dividing by zero):

```
try:
    print(1/0)
except:
    print('Exception happened!')
```

When we run this code, we get the following output:

```
Exception happened!
```

However, if we had performed division by keeping any number other than zero to be the divisor, it would have simply printed the quotient. The code inside the except block will not be executed. For example:

```
try:
    print(1/1)
except:
    print('Exception happened!')
```

```
1.0
```

Thus, if the code within the try block works, Python skips over the except block. However, if the try block causes an error, Python catches the exception and the code within the except block is executed.

The Python Exception Class

To reiterate, exceptions are special objects in Python which are used to manage errors that arise due to the occurrence of exceptional internal events (Unexpected events that disturb the program flow). **Exception** is the base class for all the exceptions in Python. The built-in exceptions and the exception hierarchy in Python are well documented here.

Thus, if we know the exception which could be raised during runtime, we can handle it even better using the built-in exceptions (named exceptions). In our case earlier, we know the exception that would be raised was the '**ZeroDivisionError**'. Hence the better way to handle it would be:

```
try:
    print(1/0)
except ZeroDivisionError:
    print('You are trying to divide a number by 0! This is not allowed!')
```

```
You are trying to divide a number by 0! This is not allowed!
```

Simply using the **except** keyword will catch any exception that is raised; By using built-in exceptions (named exceptions) we can handle each type of exception separately. It also allows us to access attributes of the exception:

```
try:
    print(1/0)
except ZeroDivisionError as e:
    print(e.args[0])
```

```
division by zero
```

Since ZeroDivisionError is derived from the **Exception** class, we could have also handled the exception in a separate manner by using the word "Exception" directly.

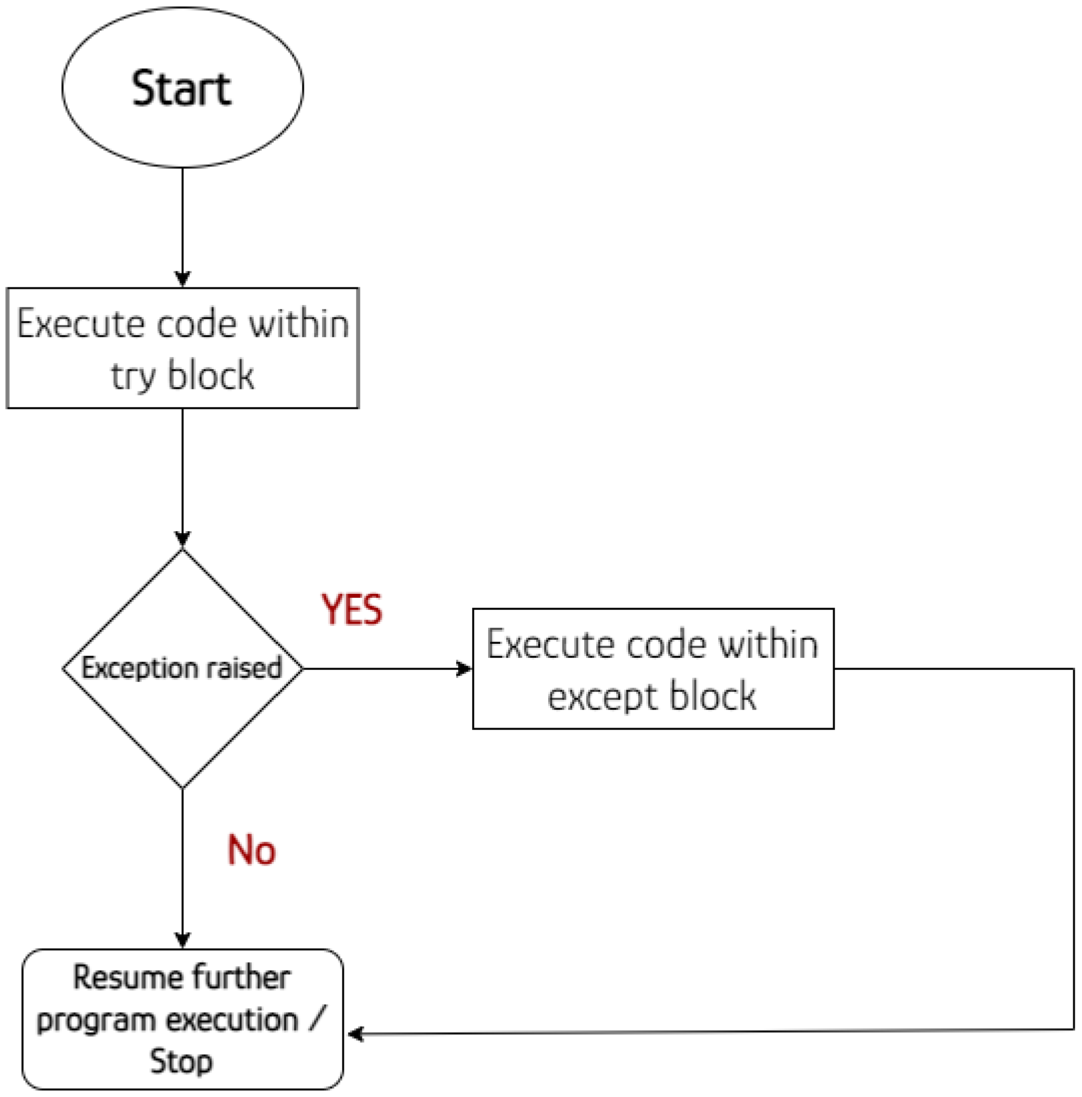
```
try:
    print(1/0)
except Exception as e:
    print(e.args[0])
```

```
division by zero
```

However, using the built-in exceptions (named exceptions) discussed previously allows us to handle the specific exceptions that might be raised and is the better way of handling exceptions. We will continue using them in this article rather than using only the except keyword or simply raising the base Exception.

In simple words, if the code within the try block works, Python skips over the except block. However, if the try block causes an error, Python looks out for an except block whose error matches the raised exception. If Python finds that, then it will run the code in that block. Once the error is handled, the rest of the program works normally.

To further understand the working of the try-except block, have a look at the flowchart given below:


Start
Execute code within try block
Exception raised
YES
Execute code within except block
No
Resume further program execution / Stop

Flowchart depicting the try-except block execution process.

Using Exception Handling to Prevent Crashes with the try-except-else Blocks

In the following piece of code, we are performing integer division of two numbers. However, we aren't printing the result of the division in the try block. We are adding another block, the '**else**' block. The code in the else block is executed only if the code within the try block is run successfully.

This makes our program more structured and robust. Only the code that can cause error goes into the try block. Exception handling statements go into the except block and the code which needs to run if and only if the try block was successful goes into the else block.

Consider the example shown below.

```
n1 = 1
n2 = 0
try:
    n3 = n1//n2
except ZeroDivisionError:
    print("You tried to divide {} by Zero!".format(str(n1)))
else:
    print("{} divided by {} gives us: {}".format(str(n1), str(n2), str(n3)))
```

```
You tried to divide 1 by Zero!
```

And if instead, we choose to keep n1 and n2 both equal to **1**, we get:

```
1 divided by 1 gives us: 1
```

Using the Keyword 'finally'

As we saw previously, the else block was executed only if the try block ran successfully. However, sometimes we want to run a piece of code that should run irrespective of whether an exception was raised or not. To do this, Python provides us with the **finally** statement.

```
n1 = 1
n2 = 0
try:
    n3 = n1//n2
except ZeroDivisionError:
    print("You tried to divide {} by Zero!".format(str(n1)))
else:
    print("{} divided by {} gives us: {}".format(str(n1), str(n2), str(n3)))
finally:
    print("This program was for integer division")
```

```
You tried to divide 1 by Zero!
This program was for integer division
```

As we can see, the else block wasn't executed. However, the finally block ran successfully. It will do so even if an exception isn't raised.

Handling Multiple Exceptions

Till now, we have just seen the ZeroDivisionError exception; However, many more built-in exceptions exist. One such error is the ‘**FileNotFoundError**‘ exception which is raised if we instruct the program to look for a file that doesn’t exist within the current working directory (or the specified location).

```
file = open('test.txt')

---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-15-a8c6235fb8e1> in <module>
----> 1 file = open('test.txt')

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

Thus, sometimes we may need to handle multiple exceptions. This can be achieved by simply cascading a set of except blocks after the try block for as many exceptions that need to be handled in the manner shown below (In case of 3 exceptions):

```
try:
  # code which may cause an exception
except Exception1:
  # Exception Handling
except Exception2:
  # Exception Handling
except Exception3:
  # Exception Handling
```

In the below example, we are reading data from an external file. That file has only one piece of content: A number. We are then using that number as our divisor and trying to divide a number by the number fetched from the external file. This process may cause 2 potential exceptions; One is the ZeroDivisionError if the number in the file is 0, And another one is the FileNotFoundError if we enter the wrong file details. Hence we will account for both of these exceptions.

```
n = 5
try:
    file = open('number')
    div = int(file.read())
    result = n//div
except FileNotFoundError:
    print('You tried to open a file which doesn\'t exist in your current working
directory.')
except ZeroDivisionError:
    print('You tried to divide by zero!')
else:
    file.close()
    print('{} divided by {} gives us {}'.format(str(n), str(div), str(result)))
```

If we run this code and a file ‘number.txt’ exists in our directory containing an integer, then we’ll get the output (in this case, the file contained the number 2):

```
5 divided by 2 gives us 2.5
```

If however, the file contained 0, the output of the above block would be:

```
You tried to divide by zero!
```

If instead of putting in ‘number’ as the argument for open() we provide in ‘numbe’ (any incorrect name), we’d get the following output:

```
You tried to open a file which doesn't exist in your current working directory.
```

In this way, we can account for multiple exceptions that may arise from within our try block.

Throwing Exceptions Manually – Using the ‘raise’ Keyword

We may want to throw our own custom exceptions for handling different kinds of errors. Sometimes, the error may not be related to Python but a logical error that we do not want. Hence, in such cases we can check a condition and then raise an exception if the condition is found to be true; The raised exception can then warn the user about the error.

For this, we will use the ‘**raise**‘ keyword. It allows us to throw exceptions manually. We can also add our message to describe the raised exception. Not only can we raise our own custom exceptions but also the built-in ones. For example:

```
n = int(input())
if n > 5:
    raise Exception('Cannot enter number greater than 5')
```

If the user enters a number greater than 5 as the input, it raises an exception with the following message:

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-3-31155e995f24> in <module>
      1 n = int(input())
      2 if n > 5:
----> 3     raise Exception('Cannot enter number greater than 5')

Exception: Cannot enter number greater than 5

try:
    n = int(input())
    if n == 0:
        raise ValueError
except ValueError:
    print ("Can\'t enter 0. Try again!")

Can't enter 0. Try again!
```

That’s it for today! If you’ve made it this far, you know what **Exceptions** are and you know how to use the **try-except** and **try-except-else blocks** to catch and handle one or more exceptions. Additionally, you also know about some special keywords in Python i.e. **finally** and **raise**.