Beginner Guide to Exception and Exception Handling in Python

Exception handling in Python is done by using the try except clause

billydharmawan
May 7 · 8 min read ★

Photo by Hitesh Choudhary on Unsplash

• • •

Introduction

In this tutorial, we will learn how to handle exceptions in Python by using the try except clause.

But first, what is an exception? 🤔

An exception is an error that is thrown by our code when the execution of the code results in an unexpected outcome. Normally, an exception will have an error type and an error message. Some examples are as follows.

```
ZeroDivisionError: division by zero
TypeError: must be str, not int
```

`ZeroDivisionError` and `TypeError` are the error type and the text that comes after the colon is the error message. The error message usually describes the error type.

Every good code and application handles exceptions to keep the user experience smooth and seamless. So, let's learn how to do that, shall we? ☺

. . .

Built-in Exceptions

In Python, all the built-in exceptions are derived from the `BaseException` class. The exception classes that directly inherit the `BaseException` class are: `Exception`, `GeneratorExit`, `KeyboardInterrupt` and `SystemExit`. We are going to focus on the `Exception` class in this tutorial.

In many cases, we want to define our own custom exception classes. The benefit of this is that we can be specific about the errors raised by our program by naming them in plain English that even the non-programmers can understand.

To define a custom exception class, we need to inherit from the `Exception` class and not the `BaseException` class as suggested in the official Python 3 documentation here.

Cool. Now that we have some basic understanding on exception, let's see some examples of it.

. . .

Common Exceptions

Now, we will be looking at some of the most basic and common exceptions we might have encountered while learning Python programming.

ValueError

This exception is raised when we pass in a value that is of the right type but the value is wrong. As an example, let's say we want to convert a string to an integer.

```
int("5")  // 5
```

This works fine, right? Now, watch what happens when we pass a string that cannot be converted to an integer.

```
int("five")

Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'five'
```

It makes sense now, right? In the second example, even though the type of the argument is fine, the value is incorrect.

IndexError

This exception tells us that we are trying to access an item or element of a collection (e.g. a string or a list object) by an index that is out of range.

```
some_list = [1, 2, 3, 4]
some_list[5]

Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: list index out of range
```

In this example, the maximum index of `some_list` is 3 and we're asking our program to access an element at index `[5]`. Because that index does not exist (or out of range), the `IndexError` exception is thrown.

TypeError

Unlike `ValueError`, this exception is raised when the type of the argument or object we are passing to a function or an expression is of the wrong type.

```
100 + "two hundred"

Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The error message is very clear here. It tells us that the `+` operator cannot be used on objects with type `int` and `str`.

NameError

This exception is thrown whenever we try to use a variable that does not exist or has not been defined yet.

```
a = "defined"

print(b)

Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'b' is not defined
```

Again, the error message is very clear. No need to explain this more. ☺

There are still many more common exceptions, but I hope this is enough examples already.

Now, you might be asking, what is that first couple of lines that says `Traceback (most recent call last)`? Well, let's discuss about it in the following section.

. . .

Photo by Marten Bjork on Unsplash

. . .

Traceback (or Stack Trace)

One thing worth mentioning from all the sample exceptions from the previous section is the part before the exception type and message. This part is known as the traceback or stack trace.

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
```

It is useful when our code is much more complex than the examples above, which I bet it will be, especially our application code. A traceback or stack trace essentially tells us the order of code executions from the latest (top) before the exception is thrown to the earliest (bottom).

Let's see some example here taken from Python official doc.

```
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
```

```
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
```

Here, the traceback is telling us that the last code execution that causes the `IndexError` exception is located on line 10, specifically when calling the `lumberjack()` function.

It also tells us that prior to calling the `lumberjack()` function, our program executes the `bright_side_of_death()` function from line 4.

In a real world application where our code is composed of many modules, the traceback is going to be helpful in debugging. Because it guides us what went wrong, where as well as the order of execution.

. . .

User Defined Exception

As mentioned earlier, Python allows us to define our own custom exceptions. To do this, we need to define a class that inherits the `Exception` class.

```
class MyCustomException(Exception):
    def __init__(self, code, message):
        self.code = code
        self.message = message
```

While we are on this, let's also see how we can raise an exception in our code. We will now define a function that will throw the `MyCustomException`.

```
def process_card(card_type):
    if card_type == "amex":
        raise MyCustomException("UNSUPPORTED_CARD_TYPE", "The card
type used is not currently supported.")
    else:
        return "OK"
```

Now, let's call `process_card` by passing `"amex"` as its argument so that it will throw the `MyCustomException`.

```
process_card("amex")

Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 3, in process_card
MyCustomException: ('UNSUPPORTED_CARD_TYPE', 'The card type used is
not currently supported.')
```

Perfect! So, in order to throw an exception from our code, the keyword to use is the `raise` keyword.

Alright, I hope by now you have a pretty good understanding of what an exception is, particularly in Python. Next, we are going to learn how to handle it.

. . .

Photo by Andrés Canchón on Unsplash

. . .

Exception Handling with Try Except Clause

Python provides us with the try except clause to handle exceptions that might be raised by our code. The basic anatomy of the try except clause is as follows.

```
try:
    // some codes
except:
    // what to do when the codes in try raise an exception
```

In plain English, the try except clause is basically saying, "Try to do this, except if there's an error, then do this instead".

There are a few options on what to do with the thrown exception from the `try` block. Let's discuss them.

Re-raise the exception

Let's take a look at how to write the try except statement to handle an exception by re-raising it.

First, let's define a function that takes two input arguments and returns their sum.

```
def myfunction(a, b):
    return a + b
```

Next, let's wrap it in a try except clause and pass input arguments with the wrong type so the function will raise the `TypeError` exception.

```
try:
    myfunction(100, "one hundred")
except:
    raise

Traceback (most recent call last):
  File "<input>", line 2, in <module>
  File "<input>", line 2, in myfunction
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

This is pretty much the same as how it would be if we didn't wrap the function in a try except clause.

Catch certain types of exception

Another option is to define which exception types we want to catch specifically. To do this, we need to add the exception type to the `except` block.

```
try:
    myfunction(100, "one hundred")
except TypeError:
    print("Cannot sum the variables. Please pass numbers only.")

Cannot sum the variables. Please pass numbers only.
```

Cool. It is starting to look better, isn't it?

Now, to make it even better, we can actually log or print the exception itself.

```
try:
    myfunction(100, "one hundred")
except TypeError as e:
    print(f"Cannot sum the variables. The exception was: {e}")

Cannot sum the variables. The exception was: unsupported operand
type(s) for +: 'int' and 'str'
```

Nice. ☺

Furthermore, we can catch multiple exception types in one `except` clause if we want to handle those exception types the same way.

Let's pass an undefined variable to our function so that it will raise the `NameError`. We will also modify our `except` block to catch both `TypeError` and `NameError` and process either exception type the same way.

```
try:
    myfunction(100, a)
except (TypeError, NameError) as e:
    print(f"Cannot sum the variables. The exception was {e}")

Cannot sum the variables. The exception was name 'a' is not defined
```

Note that we can add as many exception types as we want. Basically, we just need to pass a `tuple` containing the exception types like `(ExceptionType1, ExceptionType2, ExceptionType3, ..., ExceptionTypeN)`.

Usually, we will have another `except` block with no specific exception type. The purpose of this is to catch the unhandled exception types. A use case would be when making a Http request to a third party API, we might not know all of the possible exception types, however, we still want to catch and handle them.

Let's modify our try except block to include another except block that will catch the rest of the exception types.

```
try:
    myfunction(100, a)
except (TypeError, NameError) as e:
    print(f"Cannot sum the variables. The exception was {e}")
except Exception as e:
    print(f"Unhandled exception: {e}")
```

In the case where our function raises any exceptions other than `TypeError` and `NameError`, it will go to the last `except` block and prints `Unhandled exception: <unhandled exception here>`.

. . .

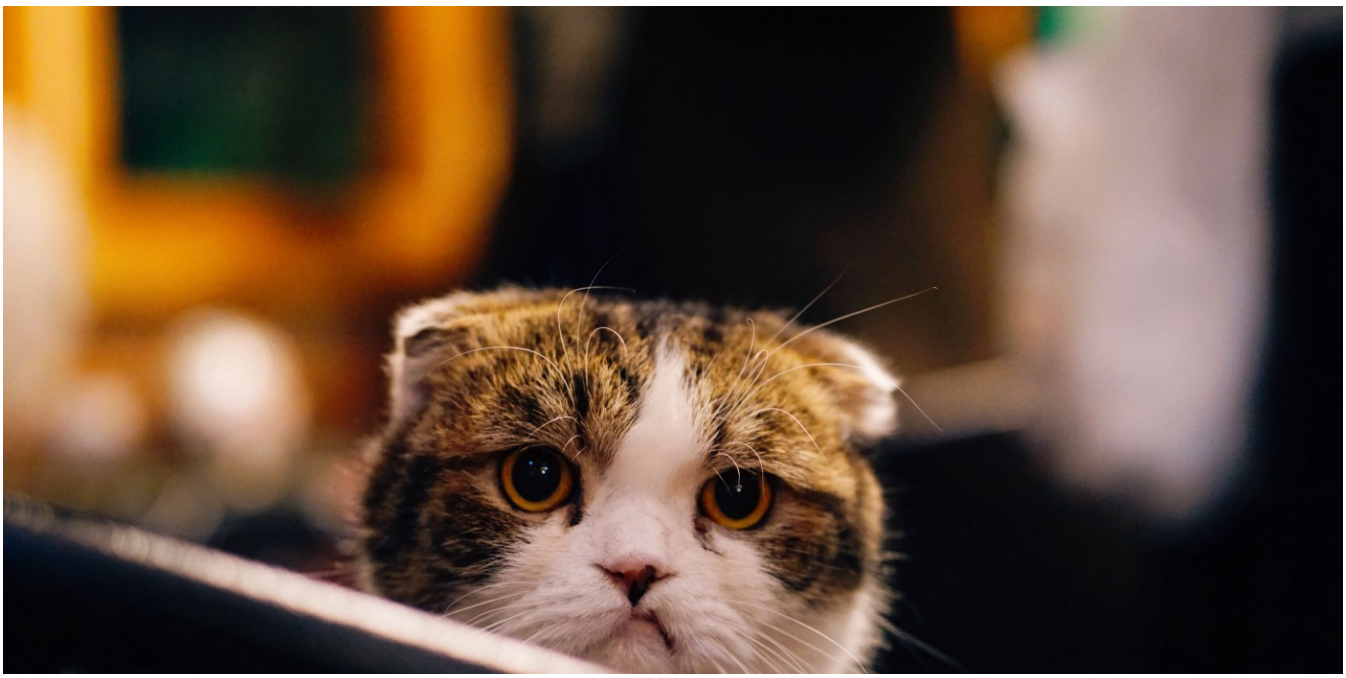

Photo by 傅甬 华 on Unsplash

. . .

Wrap Up

Wow! Great job everyone for learning about exception and exception handling in Python. I hope that wasn't too bad 😆

Just so you know, we can still extend our try except clause by adding `else` and `finally` blocks to it. More on this in the future tutorial. Or if you want to read about it yourself, you can checkout this tutorial.

As always, feel free to drop any comments or ask any questions. 🙂

Software Engineering Programming Python Coding Education