Theory: Built-in exceptions

16 minutes 0 / 5 problems solved

Skip this topic Start practicing

Even good programmers make mistakes sometimes. You can divide an integer by zero by mistake or miss a bracket when working with lists. Python handles these cases pretty well, it usually doesn't lead to unexpected bugs. But if they happen, the **built-in exceptions** are raised. In this topic, we are going to present a detailed description of built-in exceptions. Some of them (`NameError`, `TypeError`, and `ValueError`) were described in the previous topics, but there are more exceptions that we need to learn.

§1. Hierarchy of Exceptions

We should note that all built-in exceptions have a hierarchy, some of the exceptions in the hierarchy may lead to more specific exceptions. Take a look at the full structure of the built-in exceptions:

Current topic:

Built-in exceptions

Topic depends on:

✓ Exceptions Stage 1 3★

✓ Reading files Stage 2

✗ Pip Stage 1

Topic is required for:

✓ Exception handling 3★

Table of contents:

↑ Built-in exceptions

§1. Hierarchy of Exceptions

§2. Built-in Exceptions Types

§3. SyntaxError

§4. TypeError

§5. ValueError

§6. OSError

§7. ImportError

§8. EOFError

§9. NameError

§10. IndexError

§11. Conclusion

Feedback & Comments

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
```

```
        |    +-- IndentationError
        |         +-- TabError
        +-- SystemError
        +-- TypeError
        +-- ValueError
        |    +-- UnicodeError
        |         +-- UnicodeDecodeError
        |         +-- UnicodeEncodeError
        |         +-- UnicodeTranslateError
        +-- Warning
             +-- DeprecationWarning
             +-- PendingDeprecationWarning
             +-- RuntimeWarning
             +-- SyntaxWarning
             +-- UserWarning
             +-- FutureWarning
             +-- ImportWarning
             +-- UnicodeWarning
             +-- BytesWarning
             +-- ResourceWarning
```

Don't be afraid, we know, it's hard to remember the hierarchy at once. You can do it step-by-step when you have free time. Below we try to focus on the main features of the structure you need to know.

First of all, remember that the `BaseException` class is a base class for all built-in exceptions, which we can consider as the root of all exceptions. This exception is never raised on its own and should be inherited by other exception classes. In terms of this hierarchy, other classes are known as **subclasses** of the root class. For instance, the `IndexError` is a subclass of the `LookupError` class. At the same time, the `LookupError` itself is a subclass of the `Exception` class.

The `BaseException` subclasses include `SystemExit`, `KeyboardInterrupt`, `GeneratorExit` and `Exception`.

- The `SystemExit` is raised when the `sys.exit()` function is used to terminate the program.
- The `KeyboardInterrupt` is raised when the user hits the interrupt key, e.g. `Ctrl` + `C` during the execution of a program.
- The `GeneratorExit` appears when your generator is being closed (it will be covered later).
- The most common subclass is the `Exception` class. It contains all exceptions that are considered as **errors** and **warnings**.

§2. Built-in Exceptions Types

Before we start analyzing the pieces of code, take a look at the table with the built-in exceptions that programmers often meet in their code:

Exception	Explanation
`SyntaxError`	Raised when a statement uses the wrong syntax.

`TypeError`	Raised when any operation/function is applied to an object of inappropriate type.
`ValueError`	Raised when any operation/function accepts an argument with an inappropriate value.
`OSError`	Raised when a system function returns a system-related error.
`ImportError`	Raised when the imported library is not found.
`EOFError`	Raised when `input()` reaches the end of the file (EOF) without reading any data.
`NameError`	Raised when a local or global name is not found.
`IndexError`	Raised when a sequence subscript is out of range.

Now, let's shed some more light on them!

§3. SyntaxError

The first error is the `SyntaxError`. Take a look at the first example:

```
new_list = [1, 2, 3, 4, 5
print(new_list)
#   File "main.py", line 2
#     print(new_list)
#     ^
# SyntaxError: invalid syntax
```

The list doesn't have the right-side bracket `]`, that's why the `SyntaxError` is raised. Below is another example:

```
i := 3
#   File "<stdin>", line 1
#     i := 3
#        ^
# SyntaxError: invalid syntax
```

The expression is wrong and Python tells us about it by raising `SyntaxError`.

The `SyntaxError` is generally very easy to fix: you should make sure that all brackets, commas, quotation marks are in place and that everything is syntactically correct in the line that the error points out.

§4. TypeError

Now let's observe a `TypeError`:

```
a = 'Python' + 25
print(a)
# Traceback (most recent call last):
#   File "main.py", line 1, in <module>
#     a = 'Python' + 25
# TypeError: can only concatenate str (not "int") to str
```

We tried to concatenate a string and an integer, that's why the `TypeError` was raised. This can happen to you on Hyperskill. For example, if you read input and forget to convert it into an integer before performing some operations:

```
num = input()
print(num / 100)
# Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
# TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

To get rid of this error, check that you perform operations on variables of the correct data type.

§5. ValueError

In the code below, the input accepts a string that can't be converted to an integer, that's why the `ValueError` is raised:

```
user_input = input('Enter an integer: ')  # you enter 'cat' here
a = int(user_input)
print(a)
# Traceback (most recent call last):
#   File "main.py", line 1, in <module>
#    a = int(user_input)
# ValueError: invalid literal for int() with base 10: 'cat'
```

This error may also occur with a list:

```
cats = ['Tommy', 'Timmy', 'Ram']
cats.remove('Alex')
print(cats)
# Traceback (most recent call last):
#   File "main.py", line 2, in <module>
#     cats.remove('Alex')
# ValueError: list.remove(x): x not in list
```

The method `remove()` can't delete the specified string from the list because there's no such element there.

> The `ValueError` can be caused by various reasons. The general advice is to read the error message and check that the function can process the given object.

§6. OSError

The next example illustrates the `OSError`. Some subclasses of this error may appear when you work with files. For example, if we try to open a file that doesn't exist, the `FileNotFoundError` will be raised:

```
f = open('i_dont_exist.txt')
# Traceback (most recent call last):
#   File "main.py", line 1, in <module>
#     f = open('i_dont_exist.txt')

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

Of course, there are a lot of other examples when the `OSError` and its subclasses can be raised.

> When an `OSError` occurs, the reason for it is stated in the description.

§7. ImportError

The `ImportError` may occur if you import a non-existing function:

```
from math import square
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# ImportError: cannot import name 'square'
```

Or when a spelling mistake was made in the module name:

```
import maths
# Traceback (most recent call last):
#   File "main.py", line 1, in <module>
#     import maths
# ModuleNotFoundError: No module named 'maths'
```

Note that in this case, the `ModuleNotFoundError` is a subclass of the `ImportError`. Why so? The module `math` exists in the first example, but there's no such function as `square`. In Python, there's no special error subclass for this

situation, so a more general `ImportError` is raised. In the second example, however, we try to import the module that doesn't exist in Python, so the `ModuleNotFoundError` is raised.

> Apart from checking the spelling, make sure that the module you want to import is installed. If you forgot to do so, it would not be available in your program, so Python will raise this error.

§8. EOFError

Now let's discuss the `EOFError`. We have mentioned that it is raised when the input has data to read. You can run into this error on Hyperskill when, for instance, you have 2 integers as an input, one per line, but you call `input()` three times:

```
first = input()
second = input()
third = input()   # this was not expected
```

Then, the output of the tests can look like this:

```
Failed test #1 of 5. Runtime error

Error:
Traceback (most recent call last):
  File "jailed_code", line 2, in <module>
    third = input()
EOFError: EOF when reading a line
```

> You will not come across this problem often outside Hyperskill. When you get this error on Hyperskill, make sure that you read the input exactly as many times as it is stated in the task description.

§9. NameError

Take a look at the following code:

```
prant('Hello, world!')
# Traceback (most recent call last):
#   File "main.py", line 1, in <module>
#     prant('Hello, world!')
# NameError: name 'prant' is not defined
```

The function `print()` is misspelled, so Python does not recognize it. The situation is the same when you mess up the variable names:

```
a = 'Hello, world!'
print(b)
# Traceback (most recent call last):
# File "main.py", line 2, in <module>
# print(b)
# NameError: name 'b' is not defined
```

> If you get this error, just make sure that all functions and variables are correctly spelled and refer to the existing objects.

§10. IndexError

Finally, let's proceed to the `IndexError`.

```
new_list = ['the only element']
print(new_list[1])
# Traceback (most recent call last):
#   File "main.py", line 2, in <module>
#     print(new_list[1])
# IndexError: list index out of range
```

The list in the example above contains the only element, but we try to get an element with the index equal to 1. Mind that indexing in lists starts with 0. That's why the `IndexError` is raised.

> This is a very common mistake with lists. Check the indexes you are passing to your list with care and mind the number of values it has in total.

§11. Conclusion

So far, we highlighted some important issues dedicated to the built-in exceptions:

- we reviewed the hierarchy of exceptions;
- we learned what classes and subclasses stand for;
- we analyzed some built-in exceptions and discussed the way around them.

If you are keen on reading more information, check the Built-in Exceptions part of the official doc. But for now, let's proceed to the comprehension tasks and applications to strengthen your knowledge!