Python Errors and Built-in Exceptions

programiz.com/python-programming/exceptions

Join our newsletter for the latest updates.

In this tutorial, you will learn about different types of errors and exceptions that are built-in to Python. They are raised whenever the Python interpreter encounters errors.

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:

1. Syntax errors
2. Logical errors (Exceptions)

Python Syntax Errors

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

Let's look at one example:

```
>>> if a < 3
  File "<interactive input>", line 1
    if a < 3
           ^
SyntaxError: invalid syntax
```

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

We can notice here that a colon `:` is missing in the `if` statement.

Python Logical Errors (Exceptions)

Errors that occur at runtime (after passing the syntax test) are called **exceptions** or **logical errors**.

For instance, they occur when we try to open a file(for reading) that does not exist (`FileNotFoundError`), try to divide a number by zero (`ZeroDivisionError`), or 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.

Let's look at how Python treats these errors:

```
>>> 1 / 0
Traceback (most recent call last):
 File "<string>", line 301, in runcode
 File "<interactive input>", line 1, in <module>
ZeroDivisionError: division by zero

>>> open("imaginary.txt")
Traceback (most recent call last):
 File "<string>", line 301, in runcode
 File "<interactive input>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'imaginary.txt'
```

Python Built-in Exceptions

Illegal operations can raise exceptions. There are plenty of built-in exceptions in Python that are raised when corresponding errors occur. We can view all the built-in exceptions using the built-in `local()` function as follows:

```
print(dir(locals()['__builtins__']))
```

`locals()['__builtins__']` will return a module of built-in exceptions, functions, and attributes. `dir` allows us to list these attributes as strings.

Some of the common built-in exceptions in Python programming along with the error that cause them are listed below:

Exception	Cause of Error
`AssertionError`	Raised when an `assert` statement fails.
`AttributeError`	Raised when attribute assignment or reference fails.
`EOFError`	Raised when the `input()` function hits end-of-file condition.
`FloatingPointError`	Raised when a floating point operation fails.
`GeneratorExit`	Raise when a generator's `close()` method is called.
`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` or `Delete`).
`MemoryError`	Raised when an operation runs out of memory.
`NameError`	Raised when a variable is not found in local or global scope.

`NotImplementedError`	Raised by abstract methods.
`OSError`	Raised when system operation causes system related error.
`OverflowError`	Raised when the result of an arithmetic operation is too large to be represented.
`ReferenceError`	Raised when a weak reference proxy is used to access a garbage collected referent.
`RuntimeError`	Raised when an error does not fall under any other category.
`StopIteration`	Raised by `next()` function to indicate that there is no further item to be returned by iterator.
`SyntaxError`	Raised by parser when syntax error is encountered.
`IndentationError`	Raised when there is incorrect indentation.
`TabError`	Raised when indentation consists of inconsistent tabs and spaces.
`SystemError`	Raised when interpreter detects internal error.
`SystemExit`	Raised by `sys.exit()` function.
`TypeError`	Raised when a function or operation is applied to an object of incorrect type.
`UnboundLocalError`	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
`UnicodeError`	Raised when a Unicode-related encoding or decoding error occurs.
`UnicodeEncodeError`	Raised when a Unicode-related error occurs during encoding.
`UnicodeDecodeError`	Raised when a Unicode-related error occurs during decoding.
`UnicodeTranslateError`	Raised when a Unicode-related error occurs during translating.
`ValueError`	Raised when a function gets an argument of correct type but improper value.
`ZeroDivisionError`	Raised when the second operand of division or modulo operation is zero.

If required, we can also define our own exceptions in Python. To learn more about them, visit Python User-defined Exceptions.

We can handle these built-in and user-defined exceptions in Python using `try`, `except` and `finally` statements. To learn more about them, visit Python try, except and finally statements.