# Chapter 10 - Exceptions

# 10 Exceptions

Exceptions are events that occurs during the execution of a program where it disrupts the normal flow of the program's statements. You may or may not have come across some exceptions thus far after working with the various Python datatypes. Examples of exceptions that would happen can be from Lists or Tuples when a program is trying to access an invalid index, that results in an `IndexError` or trying to access a nonexistent key in a Dictionary would result in a `KeyError`. Note that errors are part of the Python Exception object and the terms are used interchangeably.

Python handles these events by raising an exception. These exceptions are generally handled by the function caller so that the program can perform a graceful shutdown otherwise the program will terminate abruptly and quit. First we will go through the types of exceptions then how to handle them.

Python has 2 types of exceptions namely:

- **Built-in Exceptions** - This uses the exceptions library provided by Python.
- **Assertions** - This uses the `assert` statement to rise an Assertion Error.

## 10.1 Built-in Exceptions

Within the built-in exceptions, there are exceptions that are caught by the parser and there are those caught by the interpreter. So what is the difference between the exceptions? The parser throws exceptions such as *Syntax Errors* and the interpreter throws exceptions that happens during the execution of a program, called *Runtime Errors*.

For instance, we can see that in following code block, line 1 has an extra bracket in the `print()` function and if we try to run this block of code, the parser will stop us and tell us that there is a Syntax Error.

```
1 print(0/0)) # there is an extra rounded bracket
2   File "main.py", line 1
3     print(0/0))
4               ^
5 SyntaxError: unmatched ')'
```

Exception information (from line 2 to 5) returned normally show us roughly where the exception has occurred (in this case, the arrow is pointing at the extra rounded bracket). Removing the extra rounded bracket and running the statement again, produces and another error but this time it is caught by the interpreter.

```
1 print(0/0)  # trying to divide by 0
2 Traceback (most recent call last):
3   File "main.py", line 1, in <module>
4     print(0/0)
5 ZeroDivisionError: division by zero
```

Errors caught by the interpreter are syntactically correct but logically wrong. This time, the exception that occurred called the `ZeroDivisionError` and it occurs when a number is being divided by zero. Python has a long list of exceptions documented here. Some common exceptions are listed in the table below

| Exception Name | Description |
|---|---|
| ZeroDivisonError | Raised when division or modulo by zero takes place for all numeric types. |
| AttributeError | Raised in case of failure of attribute reference or assignment. |
| ImportError | Raised when an import statement fails. |
| KeyboardInterrupt | Raised when the user interrupts program execution, usually by pressing Ctrl+c. |
| IndentationError | Raised when indentation is not specified properly. |
| ValueError | Raised when an operation or function receives an argument that has the right type but an inappropriate value. |
| RuntimeError | Raised when a generated error does not fall into any category. |

## 10.2 Assertions

Assertions are used as sanity-checks during development of larger programs because they are used to check the output of an expression. If the expression results in a **False**, an exception is thrown. The `assert` syntax is as follows:

```
assert expression[, Arguments]
```

Example

```
def divide(num1, num2):
    """
    Divides 2 floating point numbers
    Inputs:
            num1 - floating point number
            num2 - floating point number
    Returns:
            a floating point number
    """
    assert (num2 != 0), "The second number cannot be zero!"
    return num1/num2

print(divide(5,2))
print(divide(2,0))
```

In the example above, we are checking to see if the value of `num2` is zero. If it is, an assertion error is thrown. The first value of `2`, evaluates to a True thus the `assert` statement is not triggered but when the second value of `0` is passed, the expression results in a `False` and thus the `assert` statement is triggered.

The output of the example on the previous page is

```
2.5
Traceback (most recent call last):
  File "main.py", line 6, in <module>
    print(divide(2,0))
  File "main.py", line 2, in divide
    assert (num2 != 0), "The second number cannot be zero!"
AssertionError: The second number cannot be zero!
```

## 10.2.1 Assertions Pitfalls

There are **2 common pitfalls** of using assertions:

- Using `assert` for data validation
- `assert` that never fail

**Using `assert` for data validation**
Python assertions **can be turned off** globally using the `-O` command line option in the interpreter. This turns any `assert` statements into null operations that means the statements are not evaluated. This is an intentional design and on par with the many other programming languages.

For instance, take a look at the code below:

```
def delete_product(product_id, user):
    """
    Deletes a product from the store
    Inputs:
            product_id - string
            user - user object
    """
    assert user.is_admin(), 'Must have admin privileges to delete'
    assert store.product_exists(product_id), 'Unknown product id'
    store.find_product(product_id).delete()
```

2 serious issues can be seen:

1. **Checking for admin privileges with an assert statement is dangerous.** If the assertions are disabled in the interpreter, the `assert` statements are never evaluated thus any user is now able to delete a product. This is also considered a security leak.
2. `product_exists()` **check is skipped when assertions are disabled.** Technically we cannot delete a non existent product but in a large program, deleting any invalid product id (or information) can lead to more severe bugs down the line as we do not know how different parts of the program is developed.

To solve this we can use regular `if` statements combined with raising exceptions which we will learn at in a bit.

**`assert` that never fail**
This happens when we write `assert` statements that always evaluate to True. This happens when we try to use a `tuple` as the first argument. Remember `assert` syntax has no brackets. If brackets are used, the `assert` expression is now trying a evaluated a `tuple` but in Python, a non-empty tuple is **ALWAYS** `True`.

For instance, refer to the line of code below

```
# this will always be True because non empty tuples are always True
assert (1 == 2, 'This should fail')

# correct way of testing
assert 1 == 2, 'This definitely will fail'
```

The only solution for this is to write your code properly or use a library (such as `Pyflakes`) to check for false positives or use an assert library called `assertpy` that makes writing assertions like writing English sentences.

## 10.3 Exception Handling

Instead of letting the program terminate abruptly we can handle the exceptions using a `try...except` block of statements. Suspicious code is placed within the `try` clause and the exception is handled with the `except` clause. The general syntax is as follows:

```
try:
    suspicious codes
except ExceptionI:
    If there is ExceptionI, then execute this block.
except ExceptionII:
    If there is ExceptionII, then execute this block.
else:
    If there is no exception then execute this block.
```

Points to note:

- A single try block can handle multiple except statements. This is useful when the suspicious codes may throw different types of exceptions
- The `else` clause is optional and it is used for statements that does not require the help of the `try` block's protection

Improving on the earlier example with `try...except`.

```
def divide(num1, num2):
    """
    Divides 2 floating point numbers
    Inputs:
            num1 - floating point number
            num2 - floating point number
    Returns:
            a floating point number
    """
    result = 0
    try:
        result = num1/num2
    except ZeroDivisionError:
        print("Cannot divide by 0!")
    else:
        print("Calculation Successful")
    return result
```

```
18
19 print(divide(5,2))
20 print(divide(2,0))
```

when the code on the previous page is executed, the output is below and the program is not abruptly terminated.

```
1 Calculation Successful
2 2.5
3 Cannot divide by 0!
4 0
```

The `except` clause of the `try...except` statement can be used without defining any particular exception like so

```
1 try:
2     suspicious code
3 except:
4     If there is any Exception, then execute this block.
```

But this also means that the `except` clause will catch **all** exceptions that occur. Using this type of `try...except` statement is not considered good programming as it does not identify the root cause of the errors. But using it to identify different types of exceptions that are happening, is okay. To do that, the we need to identify the type of exception and where it was triggered. Identifying the line of code that triggered the exception requires the help of the `traceback` library. This library is normally used in conjunction with the `try...except` blocks to show exactly where the erroneous line is.

```
1  # importing the library
2  import traceback
3
4  # a loop to keep requesting for user input
5  while True:
6      try:
7          some_int = int(input("Enter a number: "))
8      except Exception as err:
9          # print the reason for the exception
10         print('Something broke: ', err)
11         # print the exception information
12         traceback.print_exc()
13         break
```

With reference to the code above, we can guess which type of inputs would trigger the exception but we do not exactly know what is the type of exception being triggered. Line 8 is similar to the `except` clauses that we have seen except for the inclusion of the `as err` part. The `as` clause is used to create an alias where by an association can be with the exception. This allows us to retrieve the message of the exception but it does not tell us where it was triggered. That is where line 12 comes in. The `print_exc()` function from the `traceback` library shows us this information. The output is as shown on the next page. More information about the `traceback` library can be found here.

```
Enter a number: f
Something broke:  invalid literal for int() with base 10: 'f'
Traceback (most recent call last):
  File "<ipython-input-5-28dc4d33e7ce>", line 7, in <module>
    some_int = int(input("Enter a number: "))
ValueError: invalid literal for int() with base 10: 'f'
```

Lastly, we have the `try...except...else...finally` block of statements. The `finally` clause is used for statements that **have to be** executed regardless of whether there are exceptions being thrown within the `try...except` block of statements or not. This `try...except...else...finally` block of statements are most commonly used for file input/output (if we are not using the `with` keyword), database operations and network socket connections, just to name a few. The general syntax is as follows

```
try:
    suspicious codes
except Exception:
    If there is Exception, then execute this block.
else:
    If there is no exception then execute this block.
finally:
    always execute this block regardless of exceptions
```

Example

```
def divide(num1, num2):
    """
    Divides 2 floating point numbers
    Inputs:
            num1 - floating point number
            num2 - floating point number
    Returns:
            a floating point number
    """
  result = 0
  try:
    result = num1/num2
  except ZeroDivisionError:
    print("Cannot divide by 0!")
  else:
    result += 5
  finally:
    result += 10
    return result

print(divide(5,2))
print(divide(2,0))
```

the output is

```
17.5
Cannot divide by 0!
10
```

Figure 1 below is a good summary on how to use the `try...except...else...finally` block of statements.

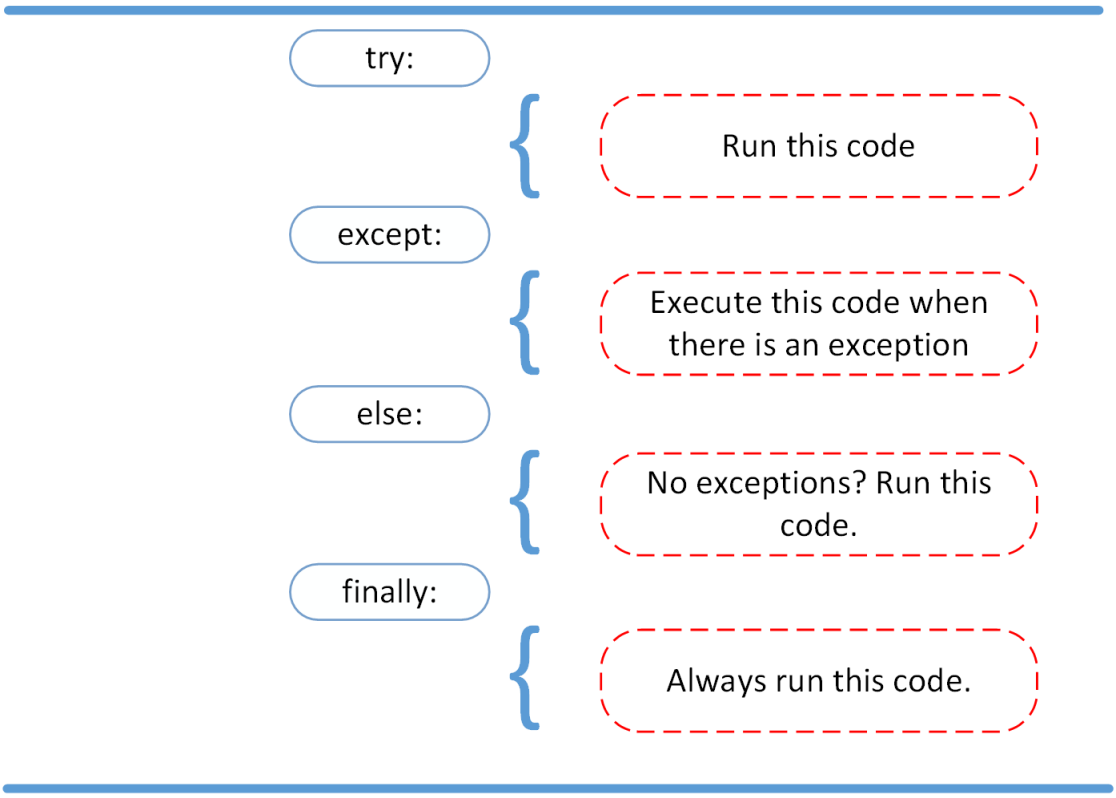


**Figure 1:** `try...except...else...finally` **block of statements summary.**

## 10.4 Raising an Exception

Instead of waiting for an Exception to occur, there is also a way to force an Exception to happen without resorting to create your own Exception library. We can do this via the `raise` statement. This differs from the `assert` statement as `assert` statements produces an `AssertionError` Exception that is used as a check for the expression but the `raise` statement is able to raise any built-in or custom created exceptions.

Raising an exception is generally done when we are aware of the conditions and are preventing it from happening so that the program will not terminate abruptly.

```
num = 10
if num > 5:
    # raising an exception
    raise Exception(f'num should not exceed 5. The value of num was: {num}')
```

In the above code block, the custom exception will be triggered when `num` is greater than `5`. The `raise` statement evaluates a given expression as an exception object, if the given of object is not an exception object, a `RuntimeError` is raised.

## 10.5 References


1. Klundert, April 2018, Python Exceptions: An Introduction, https://realpython.com/python-exceptions/
2. Python - Exceptions Handling, https://www.tutorialspoint.com/python/python_exceptions.htm
3. Lubanovic, 2019, 'Chapter 9. Functions' in Introducing Python, 2nd Edition, O`Reilly Media, Inc.
4. Built-in Exceptions, https://docs.python.org/3/library/exceptions.html
5. traceback — Print or retrieve a stack traceback, https://docs.python.org/3/library/traceback.html
6. The raise statement, https://docs.python.org/3/reference/simple_stmts.html#raise
7. Dan Bader, Assert Statements in Python, https://dbader.org/blog/python-assert-tutorial