



Python: Exception Handling and Testing

FTDS Phase 0 - Week 1 Day 4 PM

Contents



Exception and Error in Python	03
Try - Except - Else - Finally	06
Clause	09
Catching Specific Exception	10
Raising Error	11
Introduction to Program Testing	13
Python Unit Test	

Introduction to

Exception and Error in Python

Errors and exceptions in Python can cause program failures or unexpected outcomes. Python provides effective tools to enhance code stability.

```
print('halo'
```

```
File "<ipython-input-1-a4be0caa34c9>", line 1
    print('halo'
```

```
^
```

SyntaxError: incomplete input

Error Example

```
a = 2
print(2/0)
```

```
-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-2-24dd3d392f07> in <cell line: 2>()
      1 a = 2
----> 2 print(2/0)
```

ZeroDivisionError: division by zero

Exception Example

Introduction to

Exception and Error in Python

In Python, the terms "exception" and "error" are sometimes used interchangeably, but there are distinctions between them:

Exception	Error
Represents unexpected or exceptional conditions during program execution.	Indicates severe issues or unrecoverable problems that prevent program continuation.
Exceptions are caught and handled using <code>try-except</code> blocks to prevent program crashes.	Errors are usually not caught and handled using <code>try-except</code> blocks.
Exceptions can be raised explicitly using the <code>raise</code> statement or raised implicitly by the interpreter.	Errors are typically raised implicitly by the Python interpreter when an error condition occurs.
Exceptions are instances of classes derived from <code>BaseException</code> .	Errors are also instances of classes derived from <code>BaseException</code> .
Exception handling allows for graceful handling of exceptions and recovery from errors.	Errors often require code correction or termination of the program.

Introduction to Exception and Error in Python

Exception List

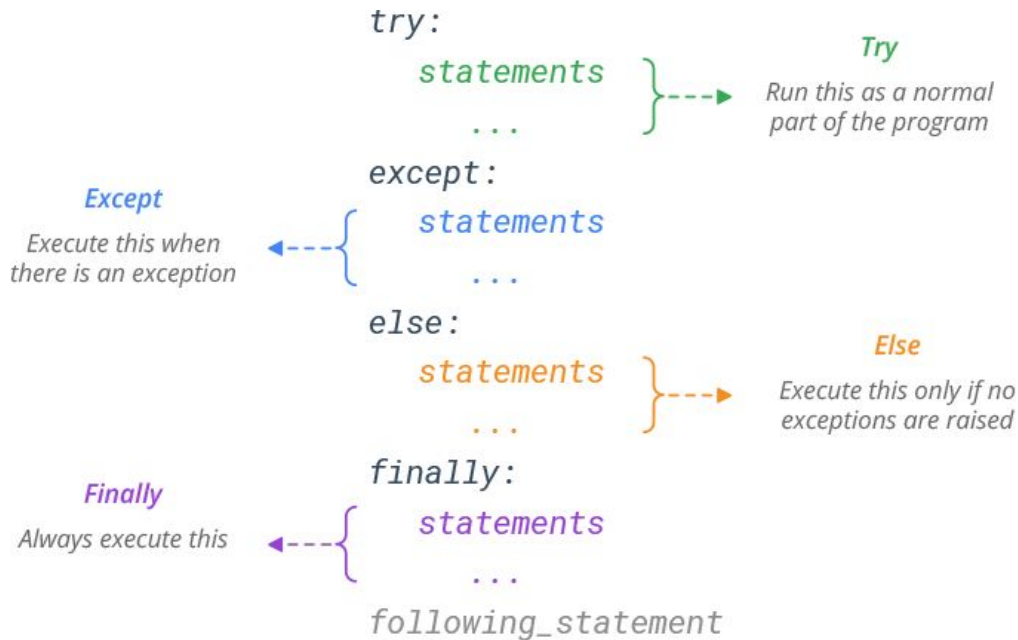
Here is the list of default Python exceptions with descriptions:

1. **AssertionError**: raised when the assert statement fails.
2. **EOFError**: raised when the input() function meets the end-of-file condition.
3. **AttributeError**: raised when the attribute assignment or reference fails.
4. **TabError**: raised when the indentations consist of inconsistent tabs or spaces.
5. **ImportError**: raised when importing the module fails.
6. **IndexError**: occurs when the index of a sequence is out of range
7. **KeyboardInterrupt**: raised when the user inputs interrupt keys (Ctrl + C or Delete).
8. **RuntimeError**: occurs when an error does not fall into any category.
9. **NameError**: raised when a variable is not found in the local or global scope.
10. **MemoryError**: raised when programs run out of memory.
11. **ValueError**: occurs when the operation or function receives an argument with the right type but the wrong value.
12. **ZeroDivisionError**: raised when you divide a value or variable with zero.
13. **SyntaxError**: raised by the parser when the Python syntax is wrong.
14. **IndentationError**: occurs when there is a wrong indentation.
15. **SystemError**: raised when the interpreter detects an internal error.

Exception and Error

Try - Except - Else - Finally Clause

We're trying to make our system can run seamlessly and without error. But, how do we do when the exception raise? We can handle it using try-except-else-finally clause.



Try - Except - Else - Finally Clause

Try - Except Example

```
for x in ["1","2","three","4"]:  
    try:  
        y = int(x)  
    except:  
        print("Cannot convert non-numerical value into integer")
```

This code is a simple example of exception handling in Python. It iterates over a list of elements: "1", "2", "three", and "4". Inside the loop, there is a `try` block where the variable `x` is attempted to be converted into an integer using the `int()` function. However, since the string "three" cannot be converted to an integer, an exception will occur.

To handle this exception, there is an `except` block that catches any exception that may occur during the conversion. In this case, the `except` block prints the message "Cannot convert non-numerical value into an integer".

So, when the code encounters the element "three", it will raise an exception, and the corresponding message will be printed. However, the code will continue executing for the remaining elements in the list.

Try - Except - Else - Finally Clause

With Else and Finally Example

```
for x in ["1","2","three","4"]:  
    try:  
        y = int(x)  
    except:  
        print("Cannot convert non-numerical value into integer")  
    else:  
        print(x)  
    finally:  
        print("This iteration is worked")
```

The Output:

```
1  
This iteration is worked  
2  
This iteration is worked  
Cannot convert non-numerical value into integer  
This iteration is worked  
4  
This iteration is worked
```


Exception and Error

Catch Specific Exception

You can catch specific exceptions by specifying the exception type in the `except` block. This allows you to handle different exceptions differently based on their specific types. Here's an example:

```
try:
    # Code that may raise exceptions
    x = 10 / 0 # This will raise a ZeroDivisionError
except ZeroDivisionError:
    # Code to handle the ZeroDivisionError
    print("Cannot divide by zero.")
except ValueError:
    # Code to handle the ValueError
    print("Invalid value.")
except Exception:
    # Code to handle any other exceptions
    print("An error occurred.")
```

Exception and Error

Raising Error

The `raise` statement allows you to indicate that a specific error condition has occurred within your code. Here's an example:

```
def divide_numbers(a, b):  
    if b == 0:  
        raise ValueError("Cannot divide by zero")  
    return a / b  
  
try:  
    result = divide_numbers(10, 0)  
    print("Result:", result)  
except ValueError as e:  
    print("Error:", str(e))
```

In this example, the **divide_numbers** function is defined to perform division between two numbers. However, before performing the division, it checks if the denominator (b) is zero. If it is, the function raises a **ValueError** using the **raise** statement, with a custom error message.

The **try-except** block is used to catch the raised exception. In this case, a **ValueError** will be raised when dividing by zero, and the corresponding **except** block will be executed. The **except** block captures the raised exception in the variable `e` and prints the error message.

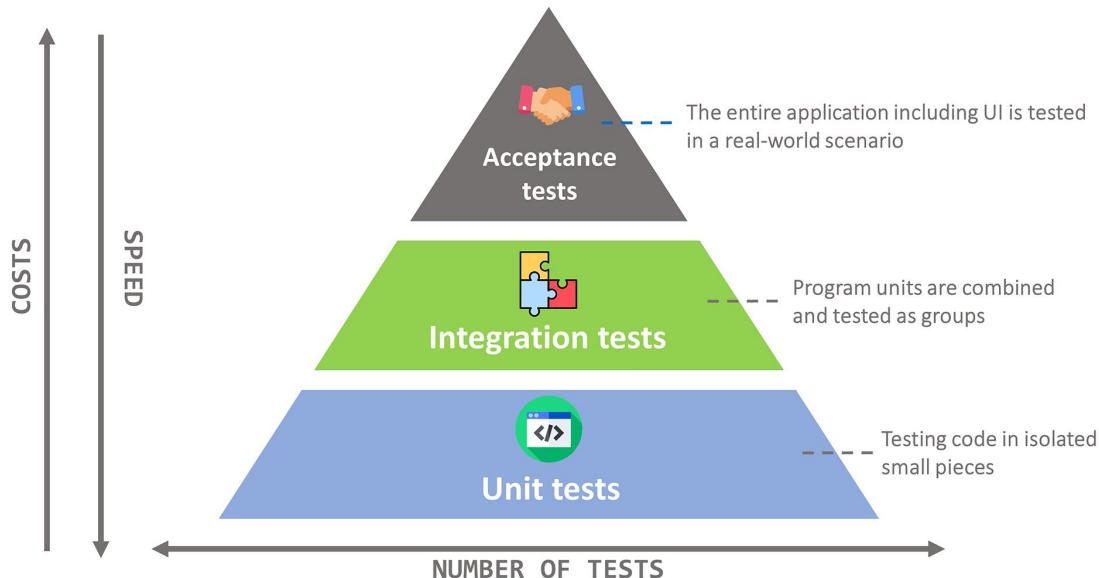
When the code is executed, the division operation encounters the zero denominator, causing a **ValueError** to be raised. The **except** block is triggered, and the error message "Cannot divide by zero" is printed.

Introduction to

Program Testing


Program testing is an important practice to verify the correctness and functionality of your code. There are three kinds of test in software development that describe in the pyramid of test.

In this case, we will focus on the **Unit Test** since this will be our scope in the industry.



Program Testing

Automated vs Manual Testing

	MANUAL TESTING	AUTOMATION TESTING
 PROCESSING TIME	Manual testing is done with the help of a human, so the time of performing this testing task depends on the amount of involved work	Automation testing is performed by a machine, program, or code, so the test is several times faster than manual testing
WAY OF TESTING	For this type of verification, a specialist test technician is used	The automatic check uses a code written in advance by a human being
LEVEL OF ROI	The ROI, in this case, is not high, as the human factor may be triggered	As the process is automated and error-free, the ROI is quite high
RELIABILITY COSTS	As the check will be carried out by a human, you will have to pay for their services	The cost of automatic checking is the cost of writing a script to test the software
CUSTOMER EXPERIENCE	Since the software will be tested by a human being, user-friendliness factors will be taken into account and the level of end-user- friendliness will be tested	The verification program does not take end-user convenience into account

In software development, you will be provided the scheme to perform testing automatically using tools.

However, all this time you write your code, you can easily test your code manually which explore the code.

But, the case of complex code, it is very time consuming. So you'll need the automation tools.

Python Unit Test

Unit Test in Python

In this course, we will perform the unit test. To perform the unit test in Python, you will need a library called **unittest**. The **unittest** has been built into the Python standard library since version 2.1. **unittest** contains both a testing framework and a test runner. unittest has some important requirements for writing and executing tests.

unittest requires that:

- You put your tests into classes as methods
- You use a series of special assertion methods in the **unittest.TestCase** class instead of the built-in **assert** statement

Python Unit Test

Example

```
import unittest

def add_numbers(a, b):
    return a + b

class TestAddNumbers(unittest.TestCase):
    def test_add_numbers(self):
        result = add_numbers(2, 3)
        self.assertEqual(result, 5)

if __name__ == '__main__':
    unittest.main()
```

In the example, we have a function **add_numbers** that takes two numbers as arguments and returns their sum. We want to write a unit test to verify that the function behaves as expected. Here's a breakdown of the code:

- The **add_numbers** function represents the functionality we want to test.
- We create a test class **TestAddNumbers** that inherits from **unittest.TestCase**, the base class for all test cases in **unittest**.
- Inside the test class, we define a test method **test_add_numbers** that verifies the behavior of the **add_numbers** function.
- In the test method, we call **add_numbers** with arguments **2** and **3** and store the result in the **result** variable.
- We use the **self.assertEqual()** method from the **TestCase** class to assert that the **result** is equal to **5**.

Python Unit Test

Example

```
$ python test_add_numbers.py
```

Output:

```
.
```

```
-----
```

```
Ran 1 test in 0.001s
```

```
OK
```

To run the unit test code, you can save it in a Python file, for example, **test_add_numbers.py**.

For the output, the dot (.) represents a successful test case. It indicates that the test **test_add_numbers** passed successfully without any failures or errors.

The output also displays information about the number of tests that were run (1 test) and the time taken to run the tests (0.001s in this case).

The OK at the end signifies that all the test cases ran successfully without any failures or errors.

Python Unit Test

Assertion

Assertions in **unittest** are used to verify expected conditions within test cases. They allow you to assert that a particular condition is true, and if it's not, the test will fail. The unittest module provides various assertion methods that can be used within test cases. Here are some commonly used assertion methods:

- **assertEqual(a, b)**: Asserts that a is equal to b.
- **assertNotEqual(a, b)**: Asserts that a is not equal to b.
- **assertTrue(x)**: Asserts that x is true.
- **assertFalse(x)**: Asserts that x is false.
- **assertIs(a, b)**: Asserts that a is the same object as b.
- **assertIsNot(a, b)**: Asserts that a is not the same object as b.
- **assertIsNone(x)**: Asserts that x is None.
- **assertIsNotNone(x)**: Asserts that x is not None.
- **assertIn(a, b)**: Asserts that a is in b.
- **assertNotIn(a, b)**: Asserts that a is not in b.
- **assertRaises(exception, callable, *args, **kwargs)**: Asserts that calling callable with args and kwargs raises the specified exception.
- **assertAlmostEqual(a, b)**: Asserts that a is approximately equal to b (within a certain tolerance).
- **assertNotAlmostEqual(a, b)**: Asserts that a is not approximately equal to b (within a certain tolerance).



External References

Colab Link

[Visit Here](#)

