Module 2 - Overview

Introduction

Module Learning Outcomes

After successful completion of this module, you will be able to do the following:

1. Write code to handle exceptions that occur during runtime
2. Define and raise your own exception classes
3. Write test suites using Python's unittest module

Key questions:

* What is an exception?
* What does it mean to handle an exception?
* How does try...except work?
* What is a unit test? Why are unit tests helpful?

Explorations

Use the pages within this module to explore the following concepts:

* Exploration: [Exception handling, unit testing](https://canvas.oregonstate.edu/courses/1915078/pages/exploration-exception-handling-unit-testing) (CLO 1c, 4, MLOs 1-3)
* Video Demo: [Exception Handling, Unit Testing](https://canvas.oregonstate.edu/courses/1915078/pages/video-demo-exception-handling-unit-testing) (CLO 1c, 4, MLOs 1-3)
* Activity: [Exception handling, unit testing](https://canvas.oregonstate.edu/courses/1915078/pages/activity-exception-handling-unit-testing) (CLO 1c, 4, MLOs 1-3)
* [Module 2 exercise solutions](https://canvas.oregonstate.edu/courses/1915078/pages/module-2-exercise-solutions)

Optional Resources

* [*Think Python* Chapter 14 sections 5, 10Links to an external site.](http://greenteapress.com/thinkpython2/html/thinkpython2015.html#sec169)
* [*Think Python* Appendix ALinks to an external site.](http://greenteapress.com/thinkpython2/html/thinkpython2021.html)

Task List

Complete the following assignments and other tasks:

* Read the Exploration pages (linked to above) and do the interactive exercises on those pages.
* Complete [Assignment 2](https://canvas.oregonstate.edu/courses/1915078/assignments/9227001), which gives you practice with exception handling and unit testing (CLO 1a, 1c, 4, MLOs 1-3).
* Take [Quiz 2](https://canvas.oregonstate.edu/courses/1915078/quizzes/2859165) (CLO 1c, 4, MLOs 1-3).

# Exploration: Exception handling, unit testing

## Exception handling

There are two kinds of problems that stop your program and print an error message. The first kind is syntax errors - things like a missing parenthesis or colon, for example. Things like that prevent the Python interpreter from being able to parse the meaning of the code. When that happens, the interpreter will print out the line where it detected the problem, with a little arrow pointing to the specific place in the line that it thinks you should look at. You also get an error message that says "SyntaxError" and a brief description of the problem. For example:

SyntaxError: unexpected EOF while parsing

**Exceptions** include all of the other errors that stop your program and print an error message. For these the interpreter will print out the line where it detected the problem and print out what type of exception occurred, with a brief description of the problem. For example:

NameError: name 'phrase' is not defined

TypeError: unsupported operand type(s) for \*\* or pow(): 'str' and 'str'

ZeroDivisionError: division by zero

Instead of allowing an exception to halt a program, you can write code to **handle** the exception. In this example we'll handle the ZeroDivisionError, which is **raised** if you try to divide (or mod) by zero.

Graphical user interface, text, application

Description automatically generated

The **try** block includes the code that might raise an exception. The **except** block specifies which exception is being handled and then describes what should happen if that exception is raised. If you run the above example with a denominator of zero, then the code in the first except block will execute. You can see how if the code in the try block can cause multiple exceptions to be raised, you can have multiple except blocks to handle them.

If you run this example with a denominator of 1, it will execute the code in the second except block (because the result is too large for an int or float to hold).

The **else** block at the end is optional, but if used it must be placed after all of the except blocks. It executes only if the code in the try block doesn't cause any exceptions. This can be useful for code that directly depends on the code in the try block executing successfully. If you run this example with a denominator of 10, it will execute the else block.

There is also an optional **finally** block, which is for clean up actions that should always happen, whether or not an exception was raised. This is typically used to release a resource the program is using (for example closing a file). This functionality has largely been superseded by the **with** statement, which we'll encounter in the lesson on file handling.

### **Raising exceptions and user defined exceptions**

You can raise an exception yourself with the **raise** keyword, for example:

raise NameError

Most of the time you don't need to raise one of the built-in exceptions - you would usually be raising an exception that you've defined for your program. You define an exception as a class that inherits from **Exception**. Inheritance lets you create a class based on an existing class - you'll learn more about that in the next module. You don't need to define any data members or methods for an exception class - instead, we'll use the **pass** keyword to tell the Python interpreter that the body is empty:

class ImaginaryNumberError(Exception):  
 pass

Then if one of your functions can't finish its calculation because of an imaginary number, it can raise the exception you defined, like this:

if intermediateResult >= 0:  
 finalResult = math.sqrt(intermediateResult)  
else:  
 raise ImaginaryNumberError

Then whoever is calling your function can put the function call inside a try block and handle it with an except block.

Here's an example of using try/catch with a user defined exception for a BankAccount class:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

If you run this, the user-defined exception InsufficientFundsError will be raised because at line 7 the account only has $200, so it's not possible to withdraw $300.

## Unit testing

Unit testing refers to testing individual units of code to make sure they work correctly before integrating them with other code. There are multiple modules available for unit testing in Python, but **unittest** is the one that's part of the Python Standard Library, so that's what we'll use here.

The unittest module defines a number of **assertions** that can be used to verify whether a certain condition holds. Here's a partial list:

Table

Description automatically generated

The last two are used to compare whether two float values are very close to equal, since comparing floats for exact equality is problematic due to possible lack of precision or round-off error.

In Python some values besides True are considered "truthy", and some values besides False are considered "falsy". As one example, None is falsy, so assertFalse(None) would be a passing test. Because of this, you should usually avoid assertTrue() and assertFalse(), and instead use assertIs() to compare a value to True or False, for example, assertIs(some\_value, False).

To create a test file, first import unittest and import the functions or classes you want to test. Next create a class that inherits from unittest.TestCase. Inside that class, define your test functions, using whatever assertions are appropriate. Each test method should have a name that starts with "test". Finally, add a main function that runs "unittest.main()". For example, let's say I want to test a function named listMax() that returns the maximum value from a list. Suppose it's defined in a file named listfuncs.py. The test class might look like the following:

Graphical user interface, text, application, email

Description automatically generated

Text

Description automatically generated

The assertion methods are defined as part of the TestCase class of the unittest module. When this testing class inherits from TestCase, it inherits all those methods. This is why it calls them with "self." in front of the names of the assertions - because they are (inherited) methods of the testing class.

Now if we run this test script (in the same way you normally run Python scripts), it will run all of the tests, print out the number of tests that were run and the number of tests that failed, and give us details about the actual results versus the expected results for the tests that failed. Try changing the code so that one or more tests fail.

Here's an example of a test file that tests a BankAccount class (without the user-defined exception class from the earlier example):

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Again, try changing the code so that one or more tests fail.

Writing good unit tests for your code **even before you've written any code** can be a tremendously helpful part of your coding process (and it's satisfying to see more of your tests passing as more of your methods and classes are completed).

## Designing tests

It's important to come up with test cases that are representative of how your code could be used. You want them to be good enough that if your code passes all your tests, you will feel confident that it correctly meets all of the requirements.

How do you know what cases to test?

* Special cases: for example if your program should compute the factorial of a number, the normal rule is that you take the product of all integers from 1 to that number, but zero is a special case, since the factorial of zero is 1, so zero should definitely be one of the test cases.
* Boundary values: for example if a customer gets a special discount when they order 10 or more items, then you should check the numbers on both sides of that boundary (9 and 10) to make sure they're handled correctly.
* Path coverage: you should have test cases that check different possible paths through your logic, to make sure the different paths give correct results.
* Edge cases: extreme or unusual cases that might expose problems.
* Unit tests: when your solution consists of multiple parts that each solve a piece of the problem, it's good to test each part separately, since it's easier to debug a small part at a time.
* Integration tests: once you're satisfied that the different parts work correctly on their own, you need to make sure they work correctly together.

It's always better for you to find cases that break your code than for a customer (or employer) to do so.

## Exercises

Try these out on your computer using PyCharm:

As you work on these questions, pay attention to the process of reading a description of requirements and figuring out a specific list of tasks. This is an important skill that you can practice on these simple exercises before you try the requirements of the assignments.

1. Define an exception named OutOfRangeError. Write a function named name\_the\_number that asks the user for an integer, and if it's equal to 1, prints "one"; if it's equal to 2, prints "two", and if it's equal to 3, prints "three". If the the parameter is not one of those three values, the function should raise an OutOfRangeError. Write code that calls name\_the\_number in a try block, and handles the possible OutOfRangeError in an except block. It should handle an OutOfRangeError by printing "That's not one of the allowed values!"

2. Write a function named multiply\_3\_numbers that takes three parameters, multiplies them together, and returns the result. Now write a test file for that function that contains at least 3 tests. Remember that the parameters could be ints or floats.

Table

Description automatically generated

Table

Description automatically generated with medium confidence

Table

Description automatically generated