BYU IDAHO CSE 111 | Programming with Functions

05 Prepare: Testing Functions

During this lesson, you will learn to use a more systematic approach to developing code. Specifically, you will learn how to write test functions that automatically verify that program functions are correct. You will learn how to use a Python module named `pytest` to run your test functions, and you will learn how to read the output of `pytest` to help you find and fix mistakes in your code.

Concepts

Here are the Python programming concepts and topics that you should learn during this lesson:

1. During previous lessons, you tested your programs by running them, typing user input, reading the output, and verifying that the output was correct. This is a valid way to test a program. However, it is time consuming, tedious, and error prone. A much better way to test a program is to test its functions individually and to write test functions that *automatically* verify that the program's functions are correct.

2. In this course, you will write test functions in a Python file that is separate from your Python program. In other words, you will keep normal program code and test code in separate files.

3. In a computer program, an assertion is a statement that causes the computer to check if a comparison is true. When the computer checks the comparison, if the comparison is true, the computer will continue to execute the code in the program. However, if the comparison is false, the computer will raise an `AssertionError`, which will likely cause the program to terminate. (In lesson 10 you will learn how to write code to handle errors so that a program won't terminate when the computer raises an error.)

   A programmer writes assertions in a program to inform the computer of comparisons that must be true in order for the program to run successfully. The Python keyword to write an assertion is `assert`. Imagine a program used by a bank to track account balances, deposits, and withdrawals. A programmer might write the first few lines of the `deposit` function like this:

```
1  def deposit(amount):
2      # In order for this program to work correctly and
3      # for the bank records to be correct, we must not
4      # allow someone to deposit a zero or negative amount.
5      assert amount > 0
6        ⋮
```

   The `assert` statement at line 5 in the previous example will cause the computer to check if the *amount* is greater than zero (0). If the amount is greater than zero, the computer will continue to execute the program. However, if the *amount* is zero or less (negative), the computer will raise an `AssertionError`, which will likely cause the program to terminate.

   A programmer can write any valid Python comparison in an `assert` statement. Here are a few examples from various unrelated programs:

```
assert temperature < 0

assert len(given_name) > 0

assert balance == 0

assert school_year != "senior"
```

4. `pytest` is a third-party Python module that makes it easy to write and run test functions. There are other Python testing modules besides `pytest`, but `pytest` seems to be the easiest to use. `pytest` is not a standard Python module. It is a third-party module. This means that when you installed Python on your computer, `pytest` was not installed, and you will need to install `pytest` in order to use it. During the checkpoint of this lesson, you will use a standard Python module named `pip` to install `pytest`.

   `pytest` allows a programmer to write simple test functions that use the Python `assert` statement to verify that a function returns a correct result. For example, if we want to verify that the built-in `min` function works correctly, we could write an `assert` statement like this:

   ```
   assert min(7, -3, 0, 2) == -3
   ```

   The previous line of code will cause the computer to first call the `min` function and pass 7, -3, 0, and 2 as arguments to the `min` function. The `min` function will find the minimum value of its parameters and return that minimum value. Then the `assert` statement will compare the returned minimum value to -3. If the returned minimum value is not -3, the `assert` statement will raise an exception which will cause `pytest` to print an error message.

5. Within a computer's memory, everything (all numbers, text, sound, pictures, movies, everything) is stored using the binary number system. When executing Python code, a computer stores integers in binary in a way that exactly represents the integers. For example, a computer executing Python code stores the integer 23 as 00010111 in binary which is an exact representation of decimal 23. However, when executing Python code, a computer approximates floating point numbers (numbers with digits after the decimal place). For example, a computer stores the floating point number 23.7 as 10000011011110110011001100110011010 in binary. This binary number is actually 23.700000762939453 in decimal which is an approximation to 23.7.

   Because computers approximate floating point numbers, we must be careful when comparing floating point numbers in our test functions. It is bad practice to check if floating point numbers are equal using just the equality operator (==). The `pytest` module contains a function named `approx` to help us compare floating point numbers. For example, to test the `math.sqrt` function, we could write an `assert` statement like this:

   ```
   assert math.sqrt(5) == approx(2.24, 0.01)
   ```

   The previous `assert` statement verifies that the value returned from `math.sqrt(5)` is within 1% (0.01) of 2.24.

6. To test a function you should do the following:
   a. Write a function that is part of your normal Python program.

   b. Think about different parameter values that will cause the computer to execute all the code in your function and will possibly cause your function to fail or return an incorrect result.

   c. In a separate Python file, write a test function that calls your program function and uses an `assert` statement to *automatically* verify that the value returned from your program function is correct.

   d. Use `pytest` to run the test function.

   e. Read the output of `pytest` and use that output to help you find and fix mistakes in both your program function and test function.

Example

Here is a simple function named `cels_from_fahr` that converts a temperature in Fahrenheit to Celsius and returns the Celsius temperature. The `cels_from_fahr` function is part of a larger Python program in a file named `weather.py`:

```
1  # weather.py
2
3  def cels_from_fahr(fahr):
4      """Convert a temperature in Fahrenheit to
5      Celsius and return the Celsius temperature.
6      """
7      cels = (fahr - 32) * 5 / 9
8      return cels
```

We want to test the `cels_from_fahr` function. From the function header at line 3 in `weather.py`, we see that `cels_from_fahr` takes one parameter named *fahr*. To adequately test this function, we should call it with at least the following: a negative number, zero, and a positive number, so in a separate file named `test_weather.py` we write a test function named `test_cels_from_fahr` as follows:

```
 1  # test_weather.py
 2
 3  from weather import cels_from_fahr
 4  from pytest import approx
 5  import pytest
 6
 7  def test_cels_from_fahr():
 8      """Test the cels_from_fahr function by calling it and comparing
 9      the values it returns to the expected values. Notice this test
10      function uses pytest.approx to compare floating point numbers.
11      """
12      assert cels_from_fahr(-25) == approx(-31.76667)
13      assert cels_from_fahr(0) == approx(-17.77778)
14      assert cels_from_fahr(32) == approx(0)
15      assert cels_from_fahr(70) == approx(21.1111)
16
17  # Call the main function that is part of pytest so that
18  # the test functions in this file will start executing.
19  pytest.main(["-v", "--tb=line", "-rN", "test_weather.py"])
```

Notice in `test_weather.py` at lines 12–15 that the test function `test_cels_from_fahr` calls the program function `cels_from_fahr` four times: once with a negative number, once with zero, and twice with positive numbers. Notice also that the test function uses `assert` and `approx`.

After writing the test function, we use `pytest` to run the test function. At line 19, instead of writing a call to the `main` function as we do in program files, we write a call to `pytest.main`. In CSE 111, in all test files, we will write a call to `pytest.main` to cause `pytest` to run the test functions. When `pytest` runs the test functions, it will produce output that tells us if the tests passed or failed like this:

```
> python test_weather.py
========================== test session starts ==========================
platform win32 -- Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy-0.13.1
rootdir: C:\Users\cse111\lesson05
collected 1 item

test_weather.py::test_cels_from_fahr PASSED                       [100%]

=========================== 1 passed in 0.10s ===========================
```

As shown above, `pytest` runs the `test_cels_from_fahr` function which calls the `cels_from_fahr` function four times and verifies that `cels_from_fahr` returns the correct value each time. We can see from the output of `pytest`, "[100%] and 1 passed", that the `cels_from_fahr` function returned the expected (correct) result all four times.

Separating Program Code from Test Code

As explained in concept #2 above, in CSE 111, we will write test functions in a separate file from program functions. It is a good idea to separate test functions and program functions because the separation makes it easy to release a program to users without releasing the test functions to them. (In general, users of a program don't want the test functions.) One consequence of writing program functions and test functions in separate files is that we must add an import statement at the top of the test file that imports all the program functions that will be tested.

Line 3 from `test_weather.py` above is an example of an import statement that imports functions from a program file. Line 3 matches this template:

```
from file_name import function_1, function_2, … function_N
```

When the computer imports functions from a file, the computer immediately executes all statements that are not written inside a function. This includes the statement to call the `main` function:

```
# Start this program by
# calling the main function.
main()
```

This means that when we run our test functions, the computer will import our program functions and at the same time, will execute the call to `main()` which will start the program executing. However, we don't want the computer to execute the program while it is executing the test functions, so we have a problem. How can we get the computer to import the program functions without executing the `main` function? Fortunately, the developers of Python gave us a solution to this problem. Instead of writing the following code to start our program running:

```
# Start this program by
# calling the main function.
main()
```

We write an `if` statement above the call to `main()` like this:

```
# If this file is executed like this:
# > python program.py
# then call the main function. However, if this file is simply
# imported (e.g. into a test file), then skip the call to main.
if __name__ == "__main__":
    main()
```

Writing the `if` statement above the call to `main()` is the correct way to write code to start a program. The Python programming language guarantees that when the computer imports the program functions (in order to test them), the comparison in the `if` statement will be false, so the computer will skip the call to `main()`. At another time, when the computer executes the program (not the test functions), the comparison in the `if` statement will be true, which will cause the computer to call the `main` function and start the program.

Documentation

The official online documentation for `pytest` contains much more information about using `pytest`. The following pages are the most applicable to CSE 111:

- Create your first test
- `assert`
- `pytest.approx`
- `pytest.main`