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:

Inefficient Testing

During previous lessons, you tested your programs by running them, typing user input, reading the program's 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 separate **test functions** that *automatically* verify that the program's functions are correct.

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.

Assert Statements

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"
```

The pytest Module

`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 a test function like this:

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

In the previous function, the `assert` statement 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 value is not −3, the `assert` statement will raise an exception which will cause `pytest` to print an error message.

Comparing Floating Point Numbers

Within a computer's memory, everything (all numbers, text, sound, pictures, movies, everything) is stored using the binary number system. While executing a Python program, a computer stores integers in binary in a way that exactly represents the integers. For example, a computer stores the integer 23 as 00010111 in binary which is an exact representation of decimal 23. However, a computer approximates floating-point numbers (numbers with digits after the decimal place). For example, while executing a Python program, a computer stores the floating-point number 23.7 as binary 0100000000110111101100110011001100110011001100110011001100110011. This binary number is actually 23.699999999999999289457264224 in decimal which is an approximation to 23.7

Because computers approximate floating-point numbers, we must carefully compare them in our test functions. It is a bad practice to check if floating-point numbers are equal using just the equality operator (==). A better way to compare two floating-point numbers is to subtract them and check if their difference is small as shown at line 8 of example 4.

```
 1  # Example 4
 2
 3      # The variables e and f can be any floating-
 4      # point numbers from any calculation.
 5      e = 7.135
 6      f = 7.128
 7
 8      if abs(e - f) < 0.01:
 9          print(f"{e} and {f} are close enough so")
10          print("we'll consider them to be equal.")
11      else:
12          print(f"{e} and {f} are not close and")
13          print("therefor not equal.")
```

In example 4 at line 8, if the difference between *e* and *f* is less than 0.01, the computer will consider the two numbers to be equal. The number 0.01 in the comparison at line 6 is called the tolerance. The **tolerance** is the maximum difference between two floating-point numbers that the programmer will allow and still consider the numbers to be equal.

`approx` Function

The comparison in example 4 at line 8 is a little tedious to write and read. Also, choosing the tolerance is sometimes difficult. The `pytest` module contains a function named `approx` to help us compare floating-point numbers more easily. The `approx` function* compares two floating-point numbers and returns `True` if they are equal within an appropriate tolerance.

> * `approx` is not a function. It is a Python class. Because CSE 111 students have not yet learned about classes, this document refers to `approx` as a function. For the purposes of CSE 111, it is sufficient for students to think of `approx` as a function.

The `approx` function has the following function header:

```python
def approx(expected_value, rel=None, abs=None, nan_ok=Fal
```

Notice that the last three parameters of the `approx` function have default values: `rel=None, abs=None, nan_ok=False`. Because they have default values, when we call `approx`, we're not required to pass arguments for the last three parameters. In other words in a test function, we can call `approx` like this:

```python
def test_function():
    assert actual_value == approx(expected_value)
```

If we call `approx` with just one argument, `approx` will compare the actual value and expected value and return `True` if the difference between the two values is less than one millionth of the expected value. In other words, one millionth of the expected value is the default tolerance. Sometimes this is not the right tolerance. The `approx` function has two parameters, `rel` and `abs`, that we can use to give `approx` a better tolerance to use in its comparison. For example, to test the `math.sqrt` function, we could write a test function like this:

```python
1   # Example 5
2
3   def test_sqrt():
4       assert math.sqrt(5) == approx(2.24, rel=0.01)
```

Notice the `rel` named argument in line 4 of the previous example. The `rel` named argument causes `approx` to compute the tolerance relative to the expected value. This means that the `assert` statement in the previous example causes the computer to verify that the actual value returned from `math.sqrt(5)` is within 1% (0.01) of 2.24. When a programmer uses the `rel` named argument, the `approx` function uses code similar to example 6 to determine if the actual and expected values are equal.

```python
# Example 6

    # Compute the tolerance.
    tolerance = expected_value * rel

    # Use the tolerance to determine if the actual
    # and expected values are close enough to be
    # considered equal.
    if abs(actual_value - expected_value) < tolerance:
        return True
    else:
        return False
```

From lines 4 and 9 of example 6, we learn that `approx` will return `True` if the difference between the actual value returned from `math.sqrt(5)` and the expected value is less than 0.0224 (2.24 * 0.01).

We can also use the `abs` named argument to give `approx` a tolerance. We can write a test for the `math.sqrt` function like this:

```
# Example 7

def test_sqrt():
    assert math.sqrt(5) == approx(2.24, abs=0.01)
```

Notice the `abs` named argument in line 4 of the previous example. The `abs` named argument causes the `approx` function to return `True` if the difference between the actual and expected values is less than the number in `abs` (0.01 in the previous example). This is different from the `rel` named argument which causes approx to return `True` if the difference is less than `rel * expected_value`. The `abs` named argument is simpler and easier to understand than the `rel` named argument.

How to Test a Function

To test a function you should do the following:

1. Write a function that is part of your normal Python program.
2. 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.
3. 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.

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

5. 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

Below 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`.

```python
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 at least three times with the following arguments.

- a negative number
- zero
- a positive number

In a separate file named `test_weather.py` we write a test function named `test_cels_from_fahr` as follows:

```python
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
9       comparing the values it returns to the expected values.
10      Notice this test function uses pytest.approx to compare
11      floating-point numbers.
12      """
13      assert cels_from_fahr(-25) == approx(-31.66667)
```

```
14     assert cels_from_fahr(0) == approx(-17.77778)
15     assert cels_from_fahr(32) == approx(0)
16     assert cels_from_fahr(70) == approx(21.1111)
17
18 # Call the main function that is part of pytest so that the
19 # computer will execute the test functions in this file.
20 pytest.main(["-v", "--tb=line", "-rN", __file__])
```

Notice in `test_weather.py` at lines 13–16 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 20, instead of writing a call to the `main` function, as we do in program files, we write a call to the `pytest.main` function. In CSE 111, at the bottom of all test files, we will write a call to `pytest.main` exactly as shown in line 20. This call to `pytest.main` will cause the `pytest` module to run our test functions. When `pytest` runs our 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, plug
rootdir: C:\Users\cse111\lesson05
collected 1 item

test_weather.py::test_cels_from_fahr PASSED

====================== 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`, "PASSED [100%]" and "1 passed", that the `cels_from_fahr` function returned the expected (correct) result all four times.

Separating Program Code from Test Code

In CSE 111, we will write test functions in a file separate 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_
```

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 s
# imported (e.g. into a test file), then skip the call to
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.

Which Program Functions Should We Test?

Because we are responsible computer programmers and want to ensure that all of our program functions work correctly, we would like to test all program functions. In other words, we would like to write at least one test function for each program function. However, this may not always be possible. The easiest program functions to test are the functions that have parameters and return a value. The hardest program functions to test are the functions that get user input, print results to a terminal window, or draw something to a window. During the next eight lessons in CSE 111, we will usually write one test function for each program function that is easy to test, meaning each function that does not get user input and does not print to a terminal window. This means that you won't write a test function for your program's `main` function because `main` usually gets user input and prints to a terminal window.

Video

Watch the following video that shows a BYU-Idaho faculty member writing two test functions and using pytest to run them.

Writing a Test Function (20 minutes)

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`

Summary

During this lesson, you are learning to write test functions that automatically verify that program functions are working correctly. In CSE 111, you will write test functions in a Python file that is separate from your program file. At the top of the test file, you will import the program functions. Then you will write one test function for each program function, except `main`. Within a test function, you will write `assert` statements that compare the value returned from a program function to the expected value. You will use a standard Python module named `pytest` to run your test functions. When a test fails, you will use the output of `pytest` to help you find and fix the mistakes in your code.