Overview

When it comes to testing, it can be difficult to know exactly where to start, especially as a beginner. This section of the lesson will be a helpful reference point as you continue to write code and test it to know what to do and when to do it.

Most projects will follow a similar pattern for the work flow:

1. **Define the problem that needs to be solved.** This includes identifying the Inputs, Process, and Outputs the problem expects to work with in it's solution.
2. **Create a function that can solve the problem with an IPO approach.** The function should use the inputs you defined for the problem, process those inputs, and then provide an expected output.
3. **Identify possible test cases.** This comes before writing the actual test file. Using your own critical thinking and problem solving, determine some possible inputs that your function may need to handle. When coming up with test cases, it's important to cover both simple and extreme cases to ensure your function behaves as expected.
4. **Calculate expected outputs for test cases.** Once you know what inputs you want to test, you need to determine what each of their expected outputs should be. This will involve using your own skills in algorithmic thinking to calculate the expected output for a given input.
5. **Create the test files.** With test cases determined, and expected outputs calculated, you are now ready to write your test functions in a test file. This is the step you use the `assert` keyword to verify inputs.
6. **Run the tests.** Run your tests and check to see if they pass or fail. If any tests fail, it's time to investigate.
7. **Iterate and test again.** Make any necessary changes to your functions and tests, and keep testing until everything passes.

Now that we have established an example workflow, let's look at an example of creating and testing a function that converts values from degrees celsius to degrees fahrenheit.

1. Define The Problem That Is Being Solved

For our example of converting Celsius to Fahrenheit, it's pretty simple to identify our input, process, and output.

- **Input:** A number representing the degrees Celsius.
- **Process:** Use the formula (°C * 9/5) + 32 = °F to calculate the temperature in degrees Fahrenheit.
- **Output:** The number representing the degrees Fahrenheit.

2. Create the Function

For our example, the function is once again very simple. In your future projects, the complexity and challenge of creating the function in python will be directly proportional to how well you planned your function in step 1.

Our function might look something like this:

`main.py`

```
def celsius_to_fahrenheit(celsius : float):
    fahrenheit = (celsius * (9 / 5)) + 32
    return fahrenheit
```

3. Identify Test Cases

When creating test cases, it's important to consider what kinds of inputs you might receive. This is one of the most difficult parts of computer science, because it's up to the programmer's judgement to consider what kinds of cases need to be tested in order to ensure their function works as intended all the time.

For our example, some possible cases we might want to ensure work could include:

- A positive number (20)
- A negative number (-20)
- 0
- A really large number (1000000)
- A really large negative number (-1000000)
- A decimal number (10.05)
- -40 (Celsius and Fahrenheit are the same temperature at -40)

4. Calculate Expected Output Values

After deciding on some test cases, we need to calculate what these values should output. For our example, we need to use our formula to calculate these values on our own, and then we can set them as our test cases.

For the following table, I used the temperature calculator from Google (which means I didn't use my formula). Before we create and run these tests, consider whether or not you agree with my method of calculating expected values. Should I have done something differently?

Input Value	Expected Output Value
20	68
-20	-4
0	32
1000000	1800032
-1000000	-1799968
10.5	50.9
-40	-40

5. Create The Test Files

For `pytest` to work, my file with tests in it needs to be named starting with "test_".

All of my test functions also need to start with "test_". When naming test functions, the name should describe what sort of case I am trying to test.

`test_temperatures.py`

```
# Import the function that I want to test
from main import celsius_to_fahrenheit

# Create the test functions with descriptive names
def test_positive():
    assert celsius_to_fahrenheit(20) == 68

def test_negative():
    assert celsius_to_fahrenheit(-20) == -4

def test_zero():
    assert celsius_to_fahrenheit(0) == 32

def test_large_positive():
    assert celsius_to_fahrenheit(1000000) == 1800032

def test_large_negative():
    assert celsius_to_fahrenheit(-1000000) == (-1799968)

def test_decimal():
    assert celsius_to_fahrenheit(10.5) == 50.9

def test_same_value():
    assert celsius_to_fahrenheit(-40) == -40
```

6. Run The Tests

I would recommend creating your own `main.py` and `test_temperatures.py` and running the tests yourself as well, as the results of the tests don't translate well to markdown files (the instruction files of these lessons).

Once you run the test files, you might be surprised at the result. Almost all of the tests passed! It may be surpising that all of the sample numbers we used (except the the decimal) result in integers, but they do. In this case, it was okay to use the Google temperature converter for us to get our results.

One of the tests did fail however. Our test for decimals had the following results:

Expected Result	Actual Result
`50.9`	`50.900000000000006`

This is one of those math quirks that comes from decimal numbers not playing nice with the binary of machine code. We will discuss this further in step 7.

7. Iterate and Test Again

In our case, we had one test fail because decimals can't be calculated with perfect precision, but the result was really close to our actual value.

Option 1: We could make it so that our results always get rounded to a specific decimal place in the function itself. However, this could result in not being able to calculate values within a certain precision down the line.

Option 2: We could use the `approx()` method from pytest that is intended specifically for working with decimal values in our test function. This option is safer, because we can still calculate temperature conversions to whatever precision we need.

Let's make the following changes to our `test_temperatures.py` file:

```
# Import the function that I want to test
from main import celsius_to_fahrenheit

# Import the approx() method from pytest
from pytest import approx

...
# Add the approx method surrounding our decimal value on the expected output side
def test_decimal():
    assert celsius_to_fahrenheit(10.5) == approx(50.9)
...
```

After making these small changes, all 7 tests we created now pass!