# Lecture 5

- Unit Tests
- `assert`
- `pytest`
- Testing Strings
- Organizing Tests into Folders
- Summing Up

## Unit Tests

- Up until now, you have been likely testing your own code using `print` statements.
- Alternatively, you may have been relying upon CS50 to test your code for you!
- It's most common in industry to write code to test your own programs.
- In your console window, type `code calculator.py`. Note that you may have previously coded this file in a previous lecture. In the text editor, make sure that your code appears as follows:

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))


def square(n):
    return n * n


if __name__ == "__main__":
    main()
```

  Notice that you could plausibly test the above code on your own using some obvious numbers such as 2. However, consider why you might want to create a test that ensures that the above code functions appropriately.

- Following convention, let's create a new test program by typing `code test_calculator.py` and modify your code in the text editor as follows:

```
from calculator import square


def main():
    test_square()


def test_square():
    if square(2) != 4:
        print("2 squared was not 4")
    if square(3) != 9:
        print("3 squared was not 9")


if __name__ == "__main__":
    main()
```

Notice that we are importing the `square` function from `square.py` on the first line of code. By convention, we are creating a function called `test_square`. Inside that function, we define some conditions to test.

- In the console window, type `python test_calculator.py`. You'll notice that nothing is being outputted. It could be that everything is running fine! Alternatively, it could be that our test function did not discover one of the "corner cases" that could produce an error.
- Right now, our code tests two conditions. If we wanted to test many more conditions, our test code could easily become bloated. How could we expand our test capabilities without expanding our test code?

## `assert`

- Python's `assert` command allows us to tell the compiler that something, some assertion, is true. We can apply this to our test code as follows:

```
from calculator import square


def main():
    test_square()


def test_square():
    assert square(2) == 4
    assert square(3) == 9


if __name__ == "__main__":
    main()
```

Notice that we are definitively asserting what `square(2)` and `square(3)` should equal. Our code is reduced from four test lines down to two.

- We can purposely break our calculator code by modifying it as follows:

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))


def square(n):
    return n + n


if __name__ == "__main__":
    main()
```

Notice that we have changed the * operator to a + in the square function.

- Now running `python test_square.py` in the console window, you will notice that an `AssertionError` is raised by the compiler. Essentially, this is the compiler telling us that one of our conditions was not met.
- One of the challenges that we are now facing is that our code could become even more

burdensome if we wanted to provide more descriptive error output to our users. Plausibly, we could code as follows:

```
from calculator import square


def main():
    test_square()


def test_square():
    try:
        assert square(2) == 4
    except AssertionError:
        print("2 squared is not 4")
    try:
        assert square(3) == 9
    except AssertionError:
        print("3 squared is not 9")
    try:
        assert square(-2) == 4
    except AssertionError:
        print("-2 squared is not 4")
    try:
        assert square(-3) == 9
    except AssertionError:
        print("-3 squared is not 9")
    try:
        assert square(0) == 0
    except AssertionError:
        print("0 squared is not 0")


if __name__ == "__main__":
    main()
```

Notice that running this code will produce multiple errors. However, it's not producing all the errors above. This is a good illustration that it's worth testing multiple cases such that you might catch situations where there are coding mistakes.

- The above code illustrates a major challenge: How could we make it easier to test your code without dozens of lines of code like the above?

You can learn more in Python's documentation of `assert`.

## `pytest`

- `pytest` is a third-party library that allows you to unit test your program. That is, you can test your functions within your program.
- To utilize `pytest` please type `pip install pytest` into your console window.
- Before applying `pytest` to our own program, modify your `test_calculator` function as follows:

```
from calculator import square
```

```
def test_assert():
    assert square(2) == 4
    assert square(3) == 9
    assert square(-2) == 4
    assert square(-3) == 9
    assert square(0) == 0
```

Notice how the above code asserts all the conditions that we want to test.

- `pytest` allows us to run our program directly through it, such that we can more easily view the results of our test conditions.
- In the terminal window, type `pytest test_calculator.py`. You'll immediately notice that output will be provided. Notice the red `F` near the top of the output, indicating that something in your code failed. Further, notice that the red `E` provides some hints about the errors in your `calculator.py` program. Based upon the output, you can imagine a scenario where `3 * 3` has outputted `6` instead of `9`. Based on the results of this test, we can go correct our `calculator.py` code as follows:

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))


def square(n):
    return n * n


if __name__ == "__main__":
    main()
```

Notice that we have changed the + operator to a * in the square function, returning it to a working state.

- Re-running `pytest test_calculator.py`, notice how no errors are produced. Congratulations!
- At the moment, it is not ideal that `pytest` will stop running after the first failed test. Again, let's return our `calculator.py` code back to its broken state:

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))


def square(n):
    return n + n


if __name__ == "__main__":
    main()
```

Notice that we have changed the * operator to a + in the square function, returning it to a broken state.

- To improve our test code, let's modify `test_calculator.py` to divide the code into different groups of tests:

```
from calculator import square


def test_positive():
    assert square(2) == 4
    assert square(3) == 9


def test_negative():
    assert square(-2) == 4
    assert square(-3) == 9


def test_zero():
    assert square(0) == 0
```

Notice that we have divided the same five tests into three different functions. Testing frameworks like `pytest` will run each function, even if there was a failure in one of them. Re-running `pytest test_calculator.py`, you will notice that many more errors are being displayed. More error output allows you to further explore what might be producing the problems within your code.

- Having improved our test code, return your `calculator.py` code to fully working order:

```
def main():
    x = int(input("What's x? "))
    print("x squared is", square(x))


def square(n):
    return n * n


if __name__ == "__main__":
    main()
```

Notice that we have changed the + operator to a * in the square function, returning it to a working state.

- Re-running `pytest test_calculator.py`, you will notice that no errors are found.
- Finally, we can test that our program handles exceptions. Let's modify `test_calculator.py` to do just that.

```
import pytest

from calculator import square


def test_positive():
    assert square(2) == 4
    assert square(3) == 9


def test_negative():
    assert square(-2) == 4
```

```
    assert square(-3) == 9


def test_zero():
    assert square(0) == 0


def test_str():
    with pytest.raises(TypeError):
        square("cat")
```

Notice that instead of using `assert`, we are taking advantage of a function within the `pytest` library itself called `raises` which allows you to express that you expect an error to be raised. We need to go to the top of our program and add `import pytest` and then call `pytest.raises` with the type of error we are expecting.

- Again, re-running `pytest test_calculator.py`, you will notice that no errors are found.
- In summary, it's up to you as a coder to define as many test conditions as you see fit!

You can learn more in Pytest's documentation of `pytest`.

## Testing Strings

- Going back in time, consider the following code `hello.py`:

```
def main():
    name = input("What's your name? ")
    hello(name)


def hello(to="world"):
    print("hello,", to)


if __name__ == "__main__":
    main()
```

  Notice that we may wish to test the result of the `hello` function.

- Consider the following code for `test_hello.py`:

```
from hello import hello


def test_hello():
    assert hello("David") == "hello, David"
    assert hello() == "hello, world"
```

  Looking at this code, do you think that this approach to testing will work well? Why might this test not work well? Notice that the `hello` function in `hello.py` prints something: That is, it does not return a value!

- We can change our `hello` function within `hello.py` as follows:

```
def main():
    name = input("What's your name? ")
    print(hello(name))
```

```
def hello(to="world"):
    return f"hello, {to}"


if __name__ == "__main__":
    main()
```

Notice that we changed our `hello` function to return a string. This effectively means that we can now use `pytest` to test the `hello` function.

- Running `pytest test_hello.py`, our code will pass all tests!
- As with our previous test case in this lesson, we can break out our tests separately:

```
from hello import hello


def test_default():
    assert hello() == "hello, world"


def test_argument():
    assert hello("David") == "hello, David"
```

Notice that the above code separates our test into multiple functions such that they will all run, even if an error is produced.

## Organizing Tests into Folders

- Unit testing code using multiple tests is so common that you have the ability to run a whole folder of tests with a single command.
- First, in the terminal window, execute `mkdir test` to create a folder called `test`.
- Then, to create a test within that folder, type in the terminal window `code test/test_hello.py`. Notice that `test/` instructs the terminal to create `test_hello.py` in the folder called `test`.
- In the text editor window, modify the file to include the following code:

```
from hello import hello


def test_default():
    assert hello() == "hello, world"


def test_argument():
    assert hello("David") == "hello, David"
```

Notice that we are creating a test just as we did before.

- `pytest` will not allow us to run tests as a folder simply with this file (or a whole set of files) alone without a special `__init__` file. In your terminal window, create this file by typing `code test/__init__.py`. Note the `test/` as before, as well as the double underscores on either side of `init`. Even leaving this `__init__.py` file empty, `pytest` is informed that the whole folder containing `__init__.py` has tests that can be run.

- Now, typing `pytest test` in the terminal, you can run the entire `test` folder of code.

You can learn more in Pytest's documentation of import mechanisms.

## Summing Up

Testing your code is a natural part of the programming process. Unit tests allow you to test specific aspects of your code. You can create your own programs that test your code. Alternatively, you can utilize frameworks like `pytest` to run your unit tests for you. In this lecture, you learned about...

- Unit tests
- `assert`
- `pytest`