Previous Next

Introduction to Unit Testing

Estimated Time: 1-2 hours

Before learning to test code, you should first understand a bit about code quality. As you continue your journey into programming, you're going to write a lot of code. At least one of your projects is going to grow into a big hairy mess of files and functions - it's inevitable. When you go to write a new function on top of this huge codebase, you'll want to make sure that adding new code doesn't break any old code. So how should you proceed? By writing test cases.

Sometimes called "unit testing" or "test-driven development" (TDD), writing tests for your code is standard nowadays. The basic idea is this: every piece of new code must have a test associated with it, and *every* test must pass before pushing any code to the git repo.

That is quite abstract and vague, so here's a simple example. Let's write a test for fizzbuzz, and then implement a fizzbuzz function that passes the test:

```
def fizzbuzz_test(f):
    if f(3) == "fizz" and f(5) == "buzz" and f(15) == "fizzbuzz":
        print("Success!")
    else:
        print("Nope. Try again.")
```

The function `fizzbuzz_test()` takes a function as an argument, runs it through a few tests, and then tells you if the function passed or not.

Try the test function with the following fizzbuzz implementation:

```
def fizzbuzz(n):
    ret = ""
    if not (n%3):
        ret += "fizz"
    if not (n%5):
        ret += "buzz"
    return ret or str(n)
```

If you run the test, it passes successfully:

```
>>> fizzbuzz_test(fizzbuzz)
Success!
```

However, you'll notice that you are only testing 3 points: `f(3)`, `f(5)`, and `f(15)`. You could totally write a fizzbuzz implementation like this:

```
def fizzjoke(n):
    if n == 3:
        return "fizz"
    if n == 5:
        return "buzz"
    if n == 15:
        return "fizzbuzz"
```

...which still passes the test, but it doesn't *actually* implement fizzbuzz. A better fizzbuzz test would look like this:

```
def fizzbuzz_goodtest(f):
    output = []
    for n in range(100):
        output.append(str(f(n) + "\n"))
```

```
    expected = open("fizzbuzz-output.txt", "r")
    i = 0
    for line in expected:
        if line == output[i]:
            print("Success!")
            i += 1
        else:
            print("Nope. Try Again.")
```

The file "fizzbuzz-output.txt" contains 100 lines of expected output. `fizzbuzz_goodtest()` simply compares this file to 100 executions of fizzbuzz. If it passes, it prints "Success!". Running `fizzbuzz_goodtest(fizzbuzz)` will print "Success!" to the terminal 100 times.

This is all well and good, but when testing more complicated code, you will find yourself writing lots of boilerplate code. Naturally, as programmers, we seek to automate things, and so Python ships with a module called unittest that you can use to write more concise tests.

The `unittest` module provides a standardized class for writing test cases, as well as functions for setting up and tearing down a test environment. For example, imagine you needed to test some database transactions. `unittest` would give you tools for automating setting up the database, populating it with test data, and tearing it down after the tests are complete.

A simple invoice calculator

Here is an example application that evenly divides a paycheck according to the number of hours worked for each employee. Your task will be to write a few test cases for this application using `unittest`. We'll guide you through the setup; first, save this in a new directory as `invoice_calculator.py`:

```
def divide_pay(amount, staff_hours):
    """
    Divide an invoice evenly amongst staff depending on how many hours
```

```
    worked on a project
    """
    total_hours = 0
    for person in staff_hours:
        total_hours += staff_hours[person]

    per_hour = amount / total_hours

    staff_pay = {}
    for person in staff_hours:
        pay = staff_hours[person] * per_hour
        staff_pay[person] = pay

    return staff_pay

def main():
    staff_pay = divide_pay(360.0, {"Alice": 3.0, "Bob": 3.0, "Carol": 
    for person, pay in staff_pay.items():
        print("{} should be paid ${:.2f}".format(person, pay))

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

Creating the test class

To get started, create a new file called `test_invoice_calculator.py`. In that file, add a stub for your test class:

```
import unittest

class InvoiceCalculatorTests(unittest.TestCase):
    pass

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

Here you import the `unittest` module, and create a new subclass of `unittest.TestCase` called `InvoiceCalculatorTests`. In your main block you call the `unittest.main` function, which will collect and run any tests contained in the file.

Save the file, then run it with `python3 test_invoice_calculator.py`. You should see the following output:

```
---
Ran 0 tests in 0.000s

OK
```

This tells you that you didn't run any tests (which is unsurprising, since you haven't written any yet), and that there were no test failures.

A simple first test

Now it's your turn. Write a test for a case in which Alice and Bob work for 3 hours, and Carol works for 6 hours on a $360 project. When complete, running the test using `python3 test_invoice_calculator.py` should pass successfully. You'll probably use the following functions:

- `assertEqual` from the unittest module - see docs here
- `divide_pay` from *invoice_calculator.py*

Hints and tips

Hint: You are going to call the `divide_pay` function with the above as input, and make sure that the pay is calculated correctly.

Bigger hint: The first change you have to make is to import the `divide_pay` function from the *invoice_calculator.py* file. Next, create a new test, called `test_divided_fairly`. In this method you need to call the `divide_pay` function for the above scenario. Then use the `assertEqual` method of the `TestCase` class to check that the value returned matches our expectations for what each person should be paid.

Testing the edge cases

At this point your test passes, so you know that your code works right? Well, not quite. Generally once you have tests for the basic operation of some code, you'll then want to identify any *edge cases* to make sure that they work properly too. Edge cases are places where there is something slightly unusual about the input which could affect the output. These are among the most common places to find bugs in code, and so are great things to test.

Try to add tests for some edge cases to see whether you can make your code more robust. First of all, add a test to make sure that if you enter zero hours for a person then it is handled correctly:

```
def test_zero_hour_person(self):
    pay = divide_pay(360.0, {"Alice": 3.0, "Bob": 6.0, "Carol": 0.0})
    self.assertEqual(pay, {"Alice": 120.0, "Bob": 240.0, "Carol": 0.0}
```

Try running the test. You should see that our code handles this case well already. Now how about if none of the staff have entered any hours? Presumably at this point you should throw an appropriate error to say that there is an incorrect input. Try to write a test for that:

```
def test_zero_hours_total(self):
    with self.assertRaises(ValueError):
        pay = divide_pay(360.0, {"Alice": 0.0, "Bob": 0.0, "Carol": 0.(
```

Here you construct your assertion in a slightly different way than before. Because you are expecting an error in the code, you have to catch the exception. In order to do this you can use a `with` block to assert that a `ValueError` is raised when you don't enter any hours.

Try running the tests. Your new test should fail, telling us that a `ZeroDivisionError` took place when you try to work out the `per_hour` amount. So let's make a slight change to *invoice_calculator.py* to make sure that the more appropriate `ValueError` is thrown when you have no hours entered:

```
def divide_pay(amount, staff_hours):
    """
    Divide an invoice evenly amongst staff depending on how many hours
    worked on a project
    """
    total_hours = 0
    for person in staff_hours:
        total_hours += staff_hours[person]

    if total_hours == 0:
        raise ValueError("No hours entered")

    per_hour = amount / total_hours

    staff_pay = {}
    for person in staff_hours:
        pay = staff_hours[person] * per_hour
        staff_pay[person] = pay

    return staff_pay
```

Notice how you've added the check to make sure that the `total_hours` variable isn't equal to zero. Now try running the tests again. They should all pass.

Let's continue with one more edge case. What happens if you pass an empty dictionary for the `staff_hours` variable? Hopefully this will be handled by the `ValueError` code you've just added, so add a test to make sure that this also works how you expect:

```
def test_no_people(self):
    with self.assertRaises(ValueError):
```

```
        pay = divide_pay(360.0, {})
```

Try running your tests. Hopefully they should all still pass, with your `ValueError` exception catching both scenarios where no hours are entered.

Is this everything?

In this assignment you've seen how to test the main functionality of some code, and added a series of further tests making sure that the edge cases are covered.

Have a think about whether you've covered all of the edge cases that could arise. Can you think of any more tests which you could add to help make your code more robust? Remember that as a developer you will have to both write and maintain your tests, so you will need to strike a balance over the number of tests you have for each unit of code. Be sure to discuss this with your mentor, and add any further tests that you decide are appropriate.

Why is unit testing your software a good idea?

Describe main elements of a test class.

Which function is used to run unit tests?

What is an assertion, and why do we use them in test cases?

How does the unittest module distinguish between test and non-test methods?

Can you give of an example of the type of code which would belong in a `setUp` or `tearDown` method?

☆ ☆ ☆ ☆ ☆ · Report a typo or other issue

✔ Mark as completed

Previous Next