**Python Unit Testing**

**Overview**

|  |
| --- |
| How do you know that your program works? Can you rely on yourself to write flawless code all the time? Meaning no disrespect, I would guess that’s unlikely. It’s quite easy to write correct code in Python most of the time, certainly, but chances are you will have bugs. In this chapter, we discuss the basics of testing. We give you some notes on how to let testing become one of your programming habits, and show you some useful tools for writing your tests. In addition to the testing and profiling tools of the standard library, there are other tools you can use to do code analysis and check your code if it conforms to python style guide, have a look at [**PyChecker**](https://github.com/akaihola/PyChecker) and [**flake8**](https://flake8.pycqa.org/en/latest/). |

**Test First, Code Later**

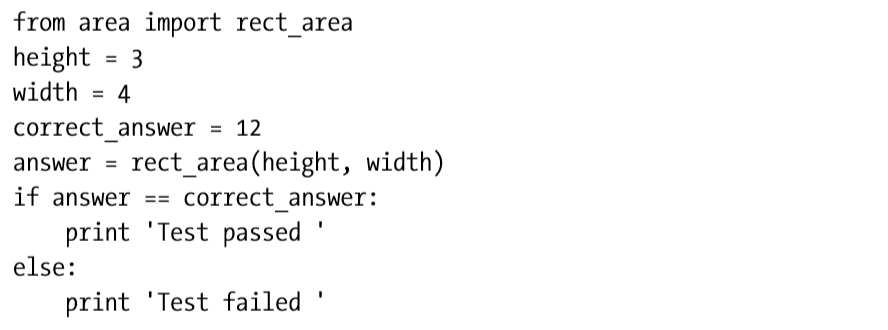
To plan for change and flexibility, which is crucial if your code is going to survive even to the end of your own development process, it’s important to set up tests for the various parts of your program (so-called ***unit tests***). Rather than the intuitive code a little, test a little practice, the Extreme programming crowd has introduced the highly useful, but somewhat counterintuitive, dictum “test a little, code a little.” In other words, test first and code later. (This is also known as **test driven development**)

**Precise Requirements Specification**

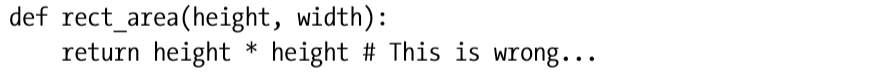
When developing a piece of software, you must first know what problem the software will solve—what objectives it will meet. You can clarify your goals for the program by writing a requirement specification, a document (or just some quick notes) describing requirements the program must satisfy. It is then easy to check at some later time whether the requirements are indeed satisfied. But many programmers dislike writing reports and in general prefer to have their computer do as much of their work as possible. Good news: You can specify the requirements in Python, and have the interpreter check whether they are satisfied!

|  |
| --- |
| **Note**: There are many types of requirements, including such vague concepts as client satisfaction. In this section, We focus on functional requirements—that is, what is required of the program’s functionality. It is very important that you follow instructions when given a task. Do not focus on the output only. It is a part of programming to ask questions when you do not understand what exactly you are expected to do. In programming we call this [**requirements analysis**](https://en.wikipedia.org/wiki/Requirements_analysis)**.** |

The idea is to start by writing a test program, and then write a program that passes the tests. The test program is your requirement specification and helps you stick to those requirements while developing the program. Let’s take a simple example: You want to write a module with a single function that will compute the area of a rectangle with a given **height** and a given **width**. Before you start coding, you write a unit test with some examples for which you know the answers. Your test program might look like the code snippets below.



In this example, we call the function ***rect\_area*** (which we haven’t written yet) on height 3 and width 4 and compare the answer with the correct one, which is 12. Of course, testing only one case like this won’t give you much confidence in the correctness of the code. A real test program would probably be a lot more thorough. Anyway, if you carelessly implement **rect\_area** (in the file **area.py**) as follows, and try to run the test program, you would get an error message:



You could then examine the code to see what was wrong, and replace the returned expression with **height \* width**.

**Planning for Change**

In addition to helping a great deal as you write the program, automated tests help you avoid accumulating errors when you introduce changes. When you change some piece of your code, you very often introduce some unforeseen bugs.

The point is that if you don’t have a thorough set of tests handy, you may not even discover that you have introduced a bug until later, when you no longer know how the error got introduced. And without a good test set, it is much more difficult to pinpoint exactly what is wrong. You can’t roll with the punches unless you see them coming. One way of making sure that you get good test coverage (that is, that your tests exercise much, if not most, of your code) is, in fact, to follow the tenets of test-driven programming. If you make sure that you have written the tests before you write the function, you can be certain that every function is tested.

**Test Driven Development Process**

Before we get into the nitty-gritty of writing tests, here’s a breakdown of the test-driven development process (or one variation of it):

1. Figure out the new feature you want. Possibly document it, and then write a test for it.
2. Write some skeleton code for the feature, so that your program runs without any syntax errors or the like, but so that your test fails. It is important to see your test fail, so you are sure that it actually can fail. If there is something wrong with the test, and it always succeeds no matter what (this has happened to me lots of times), you aren’t really testing anything. This bears repeating: See your test fail before you try to make it succeed.
3. Write a dummy code for your skeleton, just to appease the test. This doesn’t have to accurately implement the functionality, it just has to make the test pass. This way, you can have all your tests pass all the time when developing (except the first time you run the test, remember?), even while implementing the functionality initially.
4. Now you rewrite (or refactor) the code so that it actually does what it’s supposed to, all the while making sure that your test keeps succeeding.

There is a reason why on the previous task we wanted you to write two functions instead of just a few lines of code which focused on getting the output right. We introduced testing because the academy get’s feedback from the companies you will likely be interning at. It is also a good programming practice to write tests for your code. Please try by all means to follow instructions as per the task given to you.

You should keep your code in a healthy state when you leave it—don’t leave it with any tests failing. (Well, that’s what they say. I find that I sometimes leave it with one test failing, which is the point at which I’m currently working. This is really bad form if you’re developing together with others, though. You should never check failing code into the common code repository (**git**).)

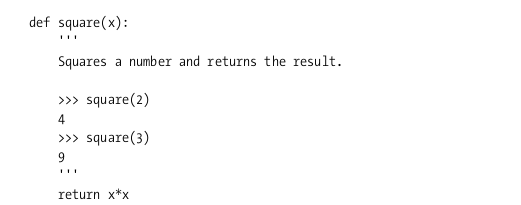
|  |
| --- |
| **Note**: Code Coverage  The concept of coverage is an important part of testing knowledge. When you run your tests, chances are you won’t run all parts of your code, even though that would be the ideal situation. One of the goals of a good test suite is to get good coverage, and one way of ensuring that is to use a coverage report tool, which measures the percent of your code that was actually run during the testing. The most important part of test-driven programming is that you actually run your method repeatedly while coding, to get continual feedback on how you’re doing. |

**Tools for Testing**

There are two brilliant modules available to automate the testing process for you: **unittest**, a python generic testing framework, and a simple module, **doctest**, which is designed for checking documentation test cases. Let’s take a look at a doctest, which is a great starting point.

**Docttest**

Let’s say I write a function for squaring a number, and add an example to its docstring like so. Try running the examples in the handouts in your pycharm or ide of your choice. **You cannot learn how to program by just reading a handout/book or by just watching a video without typing anything on your ide**:



As you can see, I’ve included some text in the **docstring**, too. You use the triple codes(‘’’ ‘’’) to add doctrings to your code.

What does this have to do with testing? Let’s say the square function is defined in the module **my\_math** (that is, a file called **my\_math.py**). Then you could add the following code at the bottom:



That’s not a lot, is it? You simply import the **doctest** and the **my\_math** module itself, and then run the **testmod** (for “test module”) function from doctest. What does this do? Let’s try it.

But before you get any visual feedback you need to give the **-v** switch (for verbose) to your script:



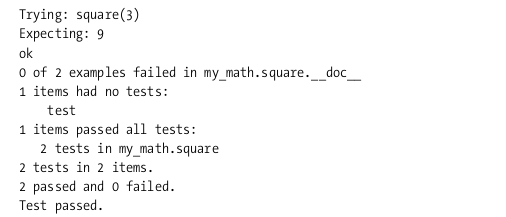
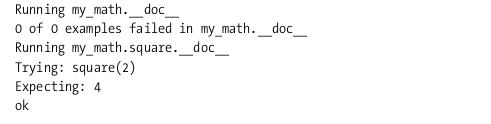
If you are using python 3, just use the following command below:

$ py my\_math.py -v

If you are using pycharm. You need to click on terminal tab as shown on the picture below:

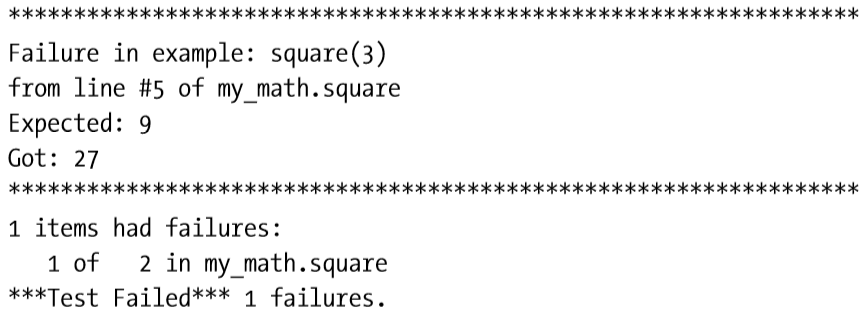


The above command will result in the following output:



As you can see, a lot happened behind the scenes. The testmod function checks both the module docstring (which, as you can see, contains no tests) and the function docstring (which contains two tests, both of which succeed)

With this in place, you can safely change your code. Let’s say that you want to use the Python exponentiation operator instead of plain multiplication, and use x\*\*2 instead of x\*x. You edit the code, but accidentally forget to enter the number 2, and end up with x\*\*x. Try it, and then run the script to test the code. What happens? This is the output you get:



So the bug was caught, and you get a very clear description of what is wrong. Fixing the problem shouldn't be difficult now.

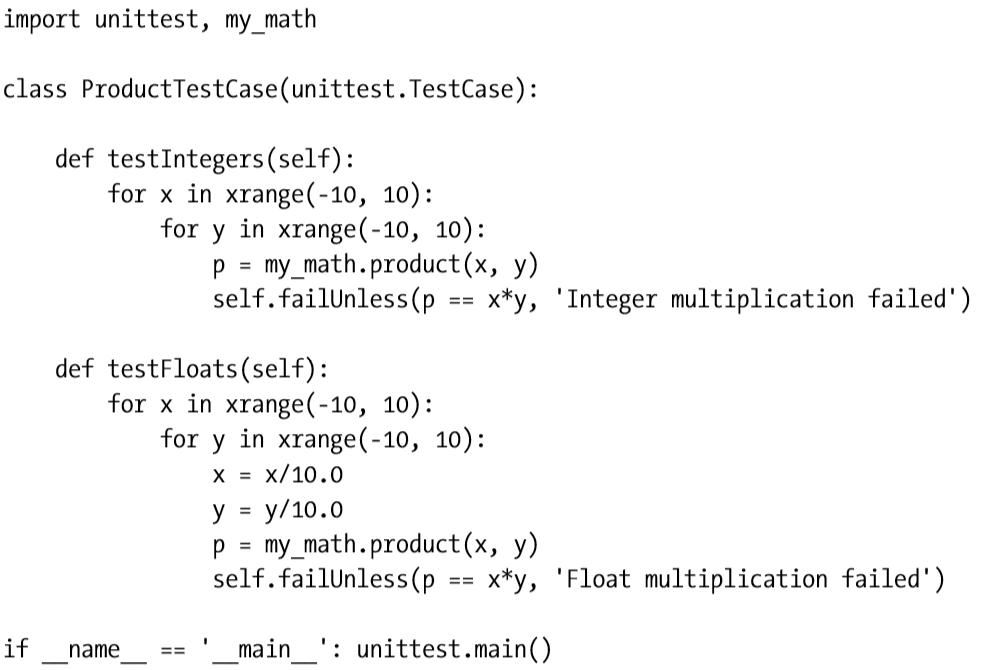
|  |
| --- |
| **Caution**: Don’t trust your tests blindly, and be sure to test enough cases. As you can see, the test using square(2) does not catch the bug because for x==2, x\*\*2 and x\*\*x are the same thing! |

For more information about the doctest module, you should again check out the library reference available on this link. (<https://docs.python.org/3/library/doctest.html>).

**Unittest**

While **doctest** is very easy to use, **unittest** (based on the popular test framework JUnit, for Java) is more flexible and powerful. It may have a steeper learning curve than doctest, but I suggest that you take a look at it because it makes it possible to write very large and thorough test sets in a more structured manner. The module is described in the library reference available on this link (<https://docs.python.org/3/library/unittest.html>).

I will only give you a gentle introduction here—there are features in the unittest you probably won’t need for most of your testing. Again, let’s take a look at a simple example. You’re going to write a module called my\_math(name you file **my\_math.py**) containing a function for calculating products, called product. So where do we begin? With a test, of course (in a file called **test\_my\_math.py**), using the TestCase class from the unittest module. Please see the code snippets below:



The function unittest.main takes care of running the tests for you. It will instantiate all subclasses of TestCase and run all methods whose names start with test.

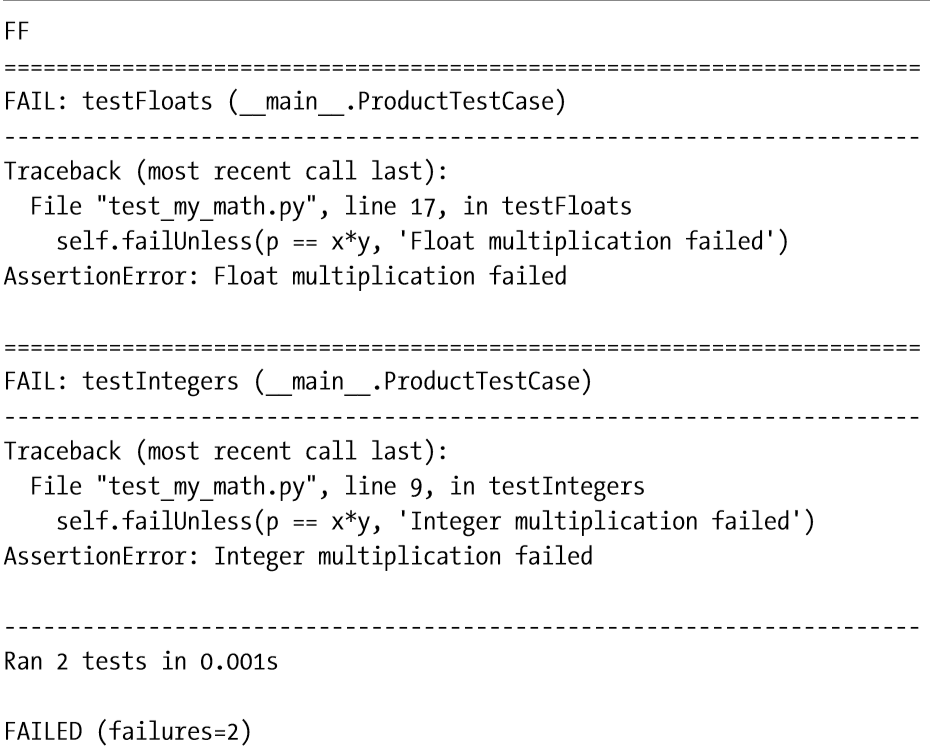
|  |
| --- |
| **Tip**: If you define methods called startUp and tearDown, they will be executed before and after each of the test methods, so you can use them to provide common initialization and cleanup code for all the tests, a so-called test fixture. |

Running this test script will, of course, simply give us an exception about the module my\_math not existing. Methods such as failUnless check a condition to determine whether the given test succeeds or fails. (There are many others, such as failIf, failUnlessEqual, failIfEqual, see the following [link](https://docs.python.org/3/library/unittest.html#unittest.TestCase.debug) for more.)

The **unittest** module distinguishes between errors, where an exception is raised, and failures, which results from calls to failUnless and the like; the next step is to write skeleton code, so we don’t get errors — only failures. This simply means to create a module called my\_math (that is, a file called my\_math.py) containing the following:



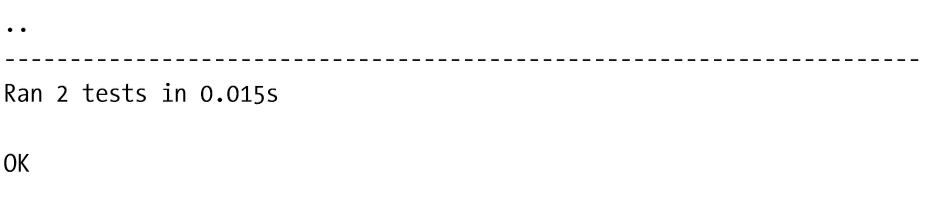
If you run the test now, you should get two **FAIL** messages, like this:



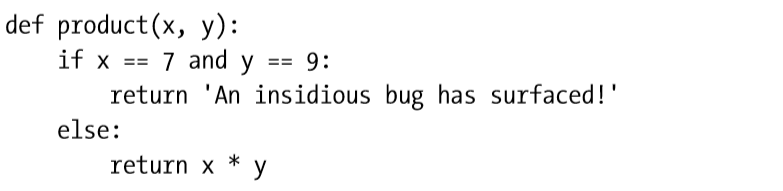
This was all expected, so don’t worry too much. Now, at least, you know that the tests are really linked to the code: The code was wrong, and the tests failed. Wonderful. Next step: Make it work. In this case, there isn’t much to if, of course:



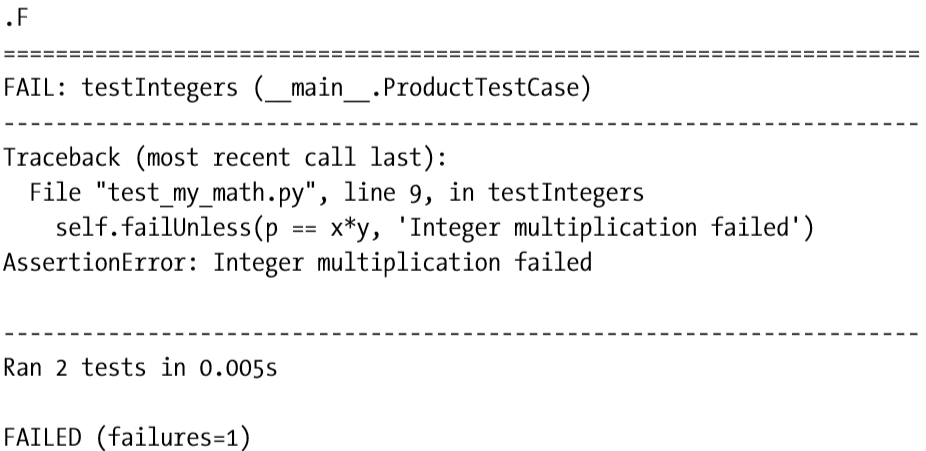
Now the output is simply



The two dots at the top are the tests. If you look closely at the jumbled output from the failed version, you’ll see that there were two characters on the top there as well: two **Fs**, indicating two failures. Just for fun, change the product function so that it fails for the specific parameters **7** and **9**:



If you run the test script again, you should get a single failure:



**Beyond Unit Tests**

Tests are clearly important, and for any somewhat complex project they are absolutely vital. Even if you don’t want to bother with structured suites of unit tests, you really have to have some way of running your program to see whether it works, and having this capability in place before you do any significant amount of coding can save you a bundle of work (and pain) later on.

**PyChecker and PyLint**

For quite some time, **PyChecker** (<http://pychecker.sf.net>) was the only tool for checking Python source code, looking for mistakes such as supplying arguments that won’t work with a given function and so forth. (All right, there was **tabnanny**, in the standard library, but that isn’t all that powerful. It just checks your indentation.) Then along came **PyLint** ([http://logilab.org/ projects/pylint](http://logilab.org/)), which supports most of the features of PyChecker, and quite a few more (such as whether your variable names fit a given naming convention, whether you’re adhering to your own coding standards, and the like).

**Continuous integration (CI)**

Continuous integration is the practice of merging in small code changes frequently - rather than merging in a large change at the end of a development cycle. The goal is to build healthier software by developing and testing in smaller increments. Visit <https://docs.travis-ci.com/user/for-beginners/#ci-builds-and-automation-building-testing-deploying> to see what happens when travis builds your project.

**A Quick Summary**

Here are the main topics covered in this module:

* **Test-driven programming**. Basically: Test first, code later. Tests let you rewrite your code with confidence, making your development and maintenance more flexible.
* **doctest and unittest.** These are indispensable tools if you want to do unit testing in Python. The doctest module is designed to check examples in docstrings, but can easily be used to design test suites. For more flexibility and structure in your suites, the unittest framework is very useful.
* **PyChecker and PyLint**. These two tools read source code and point out potential (and actual) problems. They check everything from short variable names to unreachable pieces of code. With a little coding you can make them (or one of them) part of your test suite, to make sure all of your rewrites and refactorings conform to your coding standards.
* **Travis-CI**. Is a hosted continuous integration service used to build and test software projects hosted at GitHub.
* **Profiling**. If you really care about speed and want to optimize your program (only do this if it’s absolutely necessary), you should profile it first. Use the profile (or hotshot) module to find bottlenecks in your code.