Coding Lab 6 Instructions

1 Introduction

Welcome! One of the things we want to learn in this Coding Lab is how to write *unit tests*. We will do so using the `nosetests` package, which should already be installed if you installed your packages via the `Anaconda` distribution. Head over to your terminal and type "`nosetests`". If you get something like "`command not found`", you need to install the package. Ask us if you need help with that. On the other hand, if you get something like "`Ran 0 tests in 0.001s`", then you're all set.

2 Exploring the files

Open up the file `tools.py` in your text editor. In there, you will see three silly little functions that I've written. The first one (`square`) takes in an array and squares every element of the array. The second one (`get_pi`) just returns the value of π. The last one (`picky`) does nothing but check that its input is a number.

Obviously we want to test our code! You are probably used to testing things by hand. Perhaps you print out some values for examples where you happen to know the answer. That's good, but what we really want to do is to write code to perform these tests automatically. In the directory where `tools.py` is located, type "`nosetests tests/test_tools.py`". You should get something like this:

```
adrian@~/Teaching/McGill/PHYS321/Winter2020/Labs/CodingLab06_Bayes> nosetests tests/test_tools.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```

What has happened is that we have run some automated tests on the functions found in `tools.py`. All of them passed.

Let's now see how to write these tests. Take a look in the directory `tests`, where you will find a file called `test_tools.py`. Open it up in a text editor. There are several components to this code:

- Everything exists inside a class that we have defined called `test_tools`. If you haven't used classes or done object-oriented programming before, that's ok. We won't need it very much. All you need to know is that `test_tools` defines an object that contains a bunch of functions. These functions can share variables. Any time you see "`self.{something}`", the "`self`" part refers to the object itself, and so `self.{something}` is a variable that can be accessed by any function in the object. This is why every function that is defined inside the class has "`self`" as an argument. It's so that the functions all have access to the variables (and functions) that are meant to be shared within the class.

- The function `test_square` is in charge of testing the `square` function in `tools.py`. You'll see that I wrote three different tests within this function:

  1. A test to see if the shape of the array that comes out of `square` is what I expect it to be.
  2. A test to make sure that if I input a matrix containing 1s and 0s, nothing happens to it (because $1^2 = 1$ and $0^2 = 0$).
  3. A test to make sure that for a special case that I computed by hand, `square` gives the right answer.

  In each case, notice the use of `nt.assert_equal`. This is a function that tells `nosetests` to declare that a test has failed if the two arguments are not equal.

- The function `test_get_pi` checks the output of `get_pi`. Notice the use of `nt.assert_almost_equal` here. Due to roundoff errors, the floats don't always take on the exact value that you might expect them to take on analytically, even if your code is completely correct. It is therefore prudent to allow for a little wiggle room and to require that they are "almost" equal. By default, the function assumes that "almost" means "to 7 decimal places", but that's something that you can adjust yourself using an optional argument to the function called `places`.
- The function `test_picky` checks to make sure that `picky` raises errors when it's supposed to. Recall that `picky` is only meant to accept numbers as its input. Thus, it had better raise an error if we try to feed it something that isn't a number. The function `nt.assert_raises` checks to make sure that errors are raised. In this case, we require that a `TypeError` is raised if we feed `'hey'` to `tools.picky`.
- The functions `setUp` and `tearDown` are run before and after each test, respectively. Here, you'll see for example that my `setUp` function defines a random array that can be used by any of the tests. Note that `setUp` is run before *each* test, so in this case my different tests will see different random arrays.

3 Running tests

When you typed "`nosetests tests/test_tools.py`", nosetests looked for every function beginning with `test_` and ran it. The three dots that nosetests outputted when we ran the code before each signified a test: three dots because there were three tests. Dots aren't very informative, however, so sometimes it's helpful to add `-v`, i.e., to run "`nosetests -v tests/test_tools.py`" instead. This will provide more output, for example telling you which tests passed and which did not (if any). Another helpful alternative is to add `-s`. This will output any `print` statements in the code. (Notice, for example, that in `test_square` there was a `print` statement that was ignored by `nosetests` when we didn't run with the `-s` option).

For a large code base, there can be so many functions and so many tests that it can take hours to run all the tests! Sometimes we just want to run a specific test. (For example, when we're developing a new test, we might want to just quickly try the new test out). This can be done. To run just `test_square`, for instance, I can type "`nosetests tests/test_tools.py:test_tools.test_square`".

Exercise: Try modifying the testing code to make some of the tests fail. Run the tests and see what happens! When you're done, go back to the original state of the file so that everything passes again.

4 Discussion

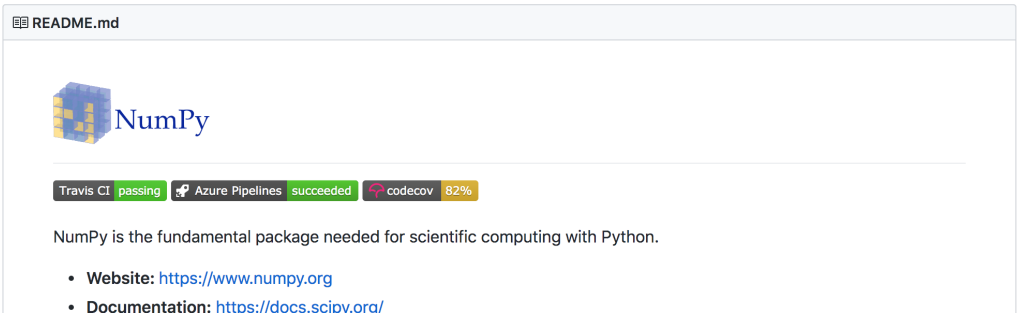
The tests that we have written here are known as *unit tests*. They are "unit" tests because they each test just a tiny chunk of the code—often the smallest chunk of code that is testable. This is in contrast to *integration testing*, for example, where we test whether the different units are working together properly.

Sometimes people say "what's the point? These tests are so simple they're silly. There's no way my functions could fail this way". My response is to ask you to ask yourself how often a bug in your code is due to some profound, subtle reason. The answer is probably "rarely". More often than not, bugs are due to silly errors that make us think "I can't believe I did that" when we find them. Unit tests are great for ironing out these problems.

Another objection that we sometimes hear is "these are the sorts of things that I do to test my code anyway. Why go to all of this trouble writing these formal tests? Doesn't that take a lot more time?" Yes, it takes some extra time, but it's worth it. Because now you can rerun all your tests with very little effort. This is important because ideally, every time we add some new functions to an existing code, we should be retesting *everything*. Usually, when we don't have prewritten test code we just test the newest functions that we have written. This is dangerous, though, because it doesn't test for the possibility that our new code accidentally

broke something we had written before. If one is doing tests by hand, it's tedious to go back and test all the old functions again. With automated testing, it's just a single command.

Ideally, every function in your code should be covered by some sort of test. Large scientific collaborations and/or large software packages often require that their code meet a minimum *test coverage* requirement. If you navigate to the `numpy`'s github repo, for example, you'll see this:



The yellow icon that says "codecov 82%" tells us that 82% of the code is covered by some sort of test. Of course, a given function can be tested in an infinite number of ways, and just because a function is "covered", it doesn't necessarily mean that we've written sensible tests that are good at finding bugs. A good workflow is to start by writing some tests that test some of the fundamental properties of a function. At some point in the code development process, you will inevitably find a bug in your code. When that happens, write a test that *would've* found that bug, and add it to your collection of tests so that such a bug will never happen again.

Although it is beyond the scope of this class, it is also possible to set up your Github repo so that every time you issue a pull request, it automatically runs all the testing code to make sure that all the tests pass. This is what the two green icons above show. It is also possible (and a good idea) to set up the repo so that new code cannot be merged into the `master` branch unless it passes all tests.

Finally, if we want to be really disciplined, a great way to write code is to do *test-driven development*. In this model, one defines the tests *before* adding new code. The tests are designed so that they pinpoint exactly what the new code is meant to do, such that they fail if the new code isn't doing its job. The tests are then written, and the test suite is run. Of course, the tests will initially fail (because the new functionality hasn't been written yet!) One then starts writing the new code, testing over and over again until all the tests have passed. At the end of the day, one has new code that has been very thoroughly tested, by construction.

5 Back to the Jupyter notebook

Head back to the Jupyter notebook, where we you will get some practice writing tests!