Package testing

December 7, 2019

Unit, integration, and system testing

Types of testing: analogy

We will describe the different types of testing using the analogy of building a car.

Consider a car that consists of four components:

1. An engine
2. An axle
3. An interior
4. A frame
5. Four wheels

Types of testing: unit testing

Unit testing involves testing a single component at a time to make sure that it behaves as you expect in isolation.

For example, in our car analogy, we might check whether the engine ran and whether pistons operated at correct frequency, etc...

What are other examples of unit testing in the context of our car?

Types of testing: integration testing

Integration testing is aimed at ensuring each part works well with the parts that it is intended to interace with.

For example, in our car analogy, we might check whether the engine was successfully able to turn the wheels through its connection on the axle.

What are other examples of integration testing in the context of our car?

Types of testing: system testing

System testing is aimed at ensuring all parts work well with one another in an environment similar to one that the software might be operated in

For example, in our car analogy, this might involve taking some of the cars that are produced for a drive in the parking lot

What are other examples of system testing in the context of our car?

Testing and project management

Testing and project management

Test-driven development

Test-driven development

Test-driven development is focused on the repitition of a very short development cycle

It begins with the idea of a particular requirement or feature and then developing with tests at the center of the (short!) development cycle

Test-driven development: Steps

Once there is a new requirement or feature that you'd like to add, the development cycle follows these steps[1]:

1. Add tests: You can consider your feature "implemented" once it can pass this set of tests
2. Run the test suite: Make sure your new tests don't already pass before you've implemented the new functionality
3. Add the new feature: Only do what is necessary here to pass the *new* test
4. Run the test suite: Ensure that the code you've added passes the new test and that it doesn't cause any of the previous tests to break
5. Refactor code: Since the development in step 3 is only about getting the test to pass, you will need to think about how it fits into your software design at this stage. You may find it necessary to modify both the recently added code AND the previously written code

[1]The Wikipedia article on test-driven development is excellent!

Test-driven development: steps

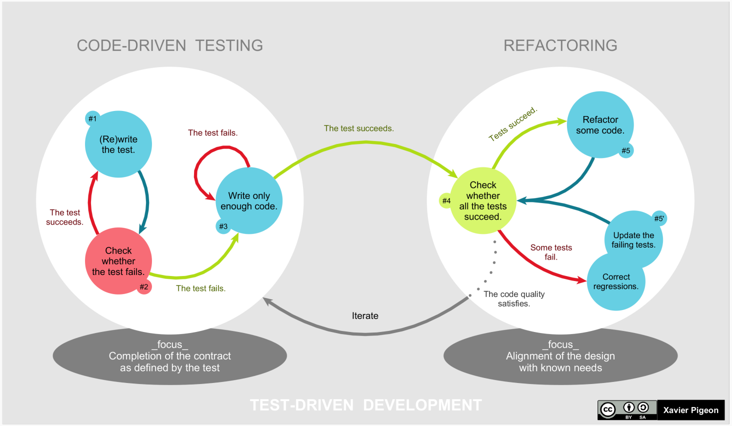

Figure attribution: By Xarawn - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=44782343

Testing in Python

Two main testing frameworks

`unittest` is a testing library that's built into the Python standard library. This means that you will always have access to this library if you're running Python.

`pytest` is a flexible and powerful testing framework. Many neat features that are worth exploring if you end up writing a relatively large test set.

unittest

We are going to focus on `unittest` today because everyone should have it installed already and it should cover our use cases for today

unittest: `unittest.TestCase`

The base component of `unittest` is the `unittest.TestCase` class

All tests are created by subclassing this class

unittest: `unittest.TestCase`

```
import unittest

class TestRandomStuff(unittest.TestCase):

    def test_addition(self):
        self.assertEqual(1 + 1, 2)

    def test_upper(self):
        self.assert('foo'.upper(), 'FOO')

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

Can run this with `python -m unittest` (can add `-v` at end if you would like verbose output)

Using unittest

Step 1: Create a folder for all of your tests — I like to use `<package>/test` but others argue for putting it closer to the source code in `<package>/<package>/test`

Step 2: Write test classes in files that begin with the word `test` — You can use a different pattern, but you have to specify it when you run the tests

Step 3: Run tests

`unittest.TestCase` features: setup and teardown

The `setUp` and `tearDown` are meant to support examples in which you require some form of temporary feature, for example, if you needed a database to run a test, then you could create a fresh database in `setUp` and delete it using `tearDown`

It's also a decent way to only define certain variables once. See this QuantEcon test

unittest.TestCase features: skip tests

```
import unittest

class TestRandomStuff(unittest.TestCase):

    @unittest.skip("Skip this test")
    def test_addition(self):
        self.assertEqual(1 + 1, 2)

    @unittest.skipUnless(sys.platform.startswith("lin"), "requires linux")
    def test_upper(self):
          self.assertEqual('foo'.upper(), 'FOO')

    @unittest.skipIf(sys.platform.startswith("win"), "Windows scares me")
    def test_something(self):
        self.assertTrue(True)

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

Exercise: write tests

Exercise: write tests!

Return to groups that built the packages yesterday — Today we're going to write tests for those packages!

Automation and continuous integration

What is continuous integration

Continuous integration is the practice of making small, but frequent, changes and updates to your code. The goal of these smaller changes is that each contribution is easier to review because it's smaller

Continuous integration tools often provide instantaneous feedback about whether new code has created issues in your build and test pipelines.

Two continuous integration providers

There are other CI providers, but the services that we'll focus on today are:

1. Travis CI
2. Github Actions

Automation and continuous integration

Travis CI

Travis CI

Travis CI is the older of the two CI services that we'll talk about today.

This means that you'll find more online help for Travis CI

Setting up Travis CI

Step 1: First, as usual, create an account on `https://travis-ci.com`

Step 2: On your account page (click the picture in the top right), you will find a list of all of your public repositories. You can turn Travis CI on by toggling the button to the right of the repository name and appropriately editing the settings

Step 3: Create and configure your `.travis.yml` file

Github actions

Github actions

Github Actions is a continuous integration system that is run by Github and is a very recent addition to their tool system. It allows you to create “workflows” which automate the software development life cycle.

You should think about this tool as being able to replace any sequence of commands that you would run each time there was a new version of your code

See Github Actions documentation

Github Actions: steps, jobs, and workflows

A *step* is the smallest size of task performed by Github Action. It consists of a single action (command)

A *job* is a defined task that consists of one or multiple steps. Each job will be run in a fresh environment. Note: Jobs can be run in parallel.

A *workflow* is a collection of one or multiple jobs. A yaml file that is kept in `.github/workflows` defines the workflow.

See Github Actions glossary

Creating a job

Exercise: automate your tests