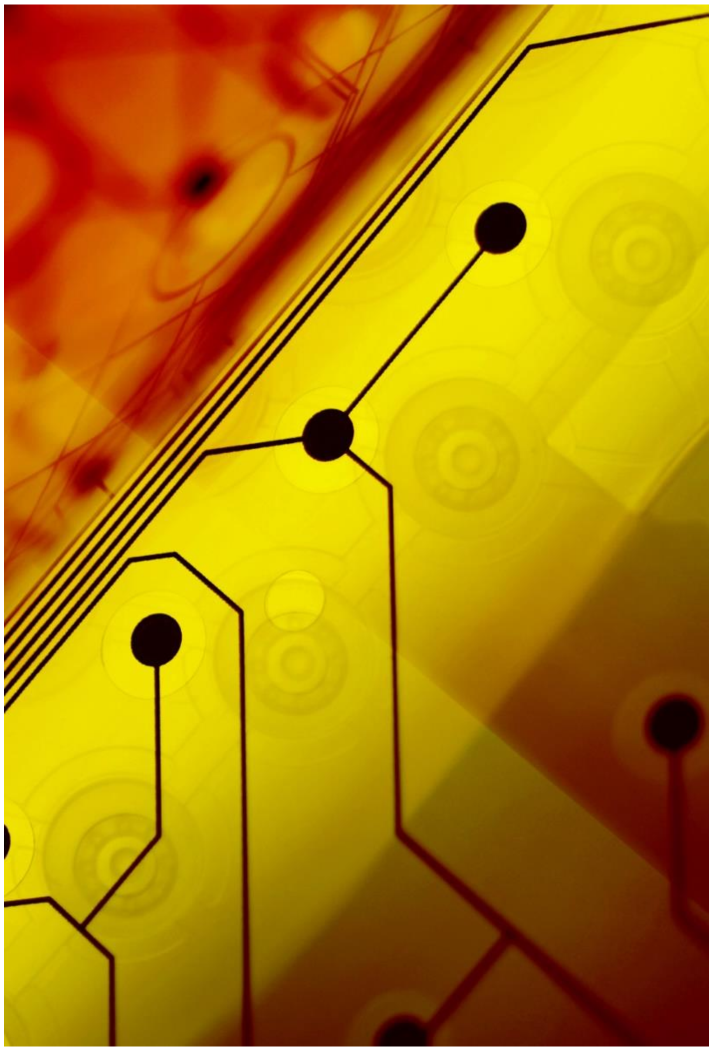


[Python Testing]

There are different models of software development. Testing is a major phase of developing software. It is important to use test plans and carry out different types of test.

[What is Unit testing?]

In computer programming, **unit testing** is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use.

[What is Integration testing?]

Integration testing is a level of software testing where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units. Test drivers and test stubs are used to assist in Integration testing.

[What is Automated testing?]

Automated testing is, well, automated. This differs from manual testing where a human being is responsible for single-handedly testing the functionality of the software in the way a user would. Because automated testing is done through an automation tool, less time is needed in exploratory tests and more time is needed in maintaining test scripts while increasing overall test coverage.

[Unit testing]

`Unittest` supports some important concepts in an object-oriented way:

test fixture

> A *test fixture* represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case

> A *test case* is the individual unit of testing. It checks for a specific response to a particular set of inputs. `unittest` provides a base class, `TestCase`, which may be used to create new test cases.

test suite

> A *test suite* is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

test runner

> A *test runner* is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

[Basic example]

The `unittest` module provides a rich set of tools for constructing and running tests. This section demonstrates that a small subset of the tools suffice to meet the needs of most users.

Here is a short script to test three string methods:

```python
import unittest

class TestStringMethods(unittest.TestCase):

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

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

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

A testcase is created by subclassing unittest.TestCase. The three individual tests are defined with methods whose names start with the letters test. This naming convention informs the test runner about which methods represent tests.

The crux of each test is a call to assertEqual() to check for an expected result; assertTrue() or assertFalse() to verify a condition; or assertRaises() to verify that a specific exception gets raised. These methods are used instead of the assert statement so the test runner can accumulate all test results and produce a report.

The setUp() and tearDown() methods allow you to define instructions that will be executed before and after each test method. They are covered in more detail in the section Organizing test code.

The final block shows a simple way to run the tests. unittest.main() provides a command-line interface to the test script. When run from the command line, the above script produces an output that looks like this:

[Integration testing]

Creating Integration Tests

In Pyramid, a *unit test* typically relies on "mock" or "dummy" implementations to give the code under test enough context to run.

"Integration testing" implies another sort of testing. In the context of a Pyramid integration test, the test logic exercises the functionality of the code under test *and* its integration with the rest of the Pyramid framework.

Creating an integration test for a Pyramid application usually means invoking the application's includeme function via pyramid.config.Configurator.include() within the test's setup code. This causes the entire Pyramidenvironment to be set up, simulating what happens when your application is run "for real". This is a heavy-hammer way of making sure that your tests have enough context to run properly, and tests your code's integration with the rest of Pyramid.

[Simple example]

As a simple example, consider the three functions a(), b(), and c(). The a() function adds one to a number, b() multiplies a number by two, and c() composes them. These functions are defined as follows:

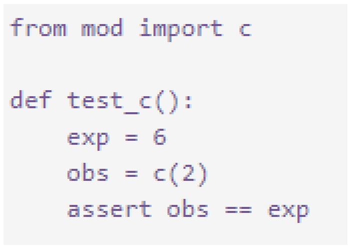
```
def a(x):
    return x + 1

def b(x):
    return 2 * x

def c(x):
    return b(a(x))
```

The a() and b() functions can each be unit tested because they each do one thing. However, c() cannot be truly unit tested because all of the real work is farmed out to a() and b(). Testing c() will be a test of whether a() and b() can be integrated together.

Integration tests still follow the pattern of comparing expected results to observed results. A sample test_c() is implemented here:



```
from mod import c

def test_c():
    exp = 6
    obs = c(2)
    assert obs == exp
```

Given the lack of clarity in what is defined as a code unit, what is considered an integration test is also a little fuzzy. Integration tests can range from the extremely simple (like the one just shown) to the very complex. A good delimiter, though, is in opposition to the unit tests. If a function or class only combines two or more unit-tested pieces of code, then you need an integration test. If a function implements new behavior that is not otherwise tested, you need a unit test.

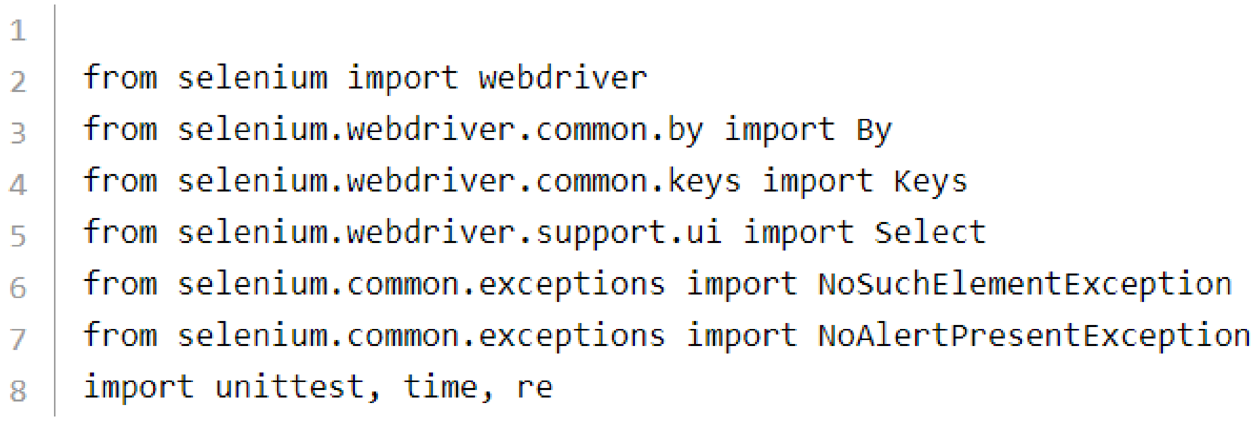
The structure of integration tests is very similar to that of unit tests. There is an expected result, which is compared against the observed value. However, what goes in to creating the expected result or setting up the code to run can be considerably more complicated and more involved. Integration tests can also take much longer to run because of how much more work they do. This is a useful classification to keep in mind while writing tests. It helps separate out which test should be easy to write (unit) and which ones may require more careful consideration (integration).

[Automated testing]

Automated testing is a powerful concept that allows you - by using separate software - to perform self-acting tests and compare their actual outcome with expected results. This kind of tests can automate some repetitive, low-level UI checks, but this approach also works perfectly with more sophisticated end-to-end tests.

Automated tests have several advantages over manual testing. One of the greatest things is that once they have been developed, they can be run quickly and repeatedly. They can also save a lot of money in the long term because more tests can be performed in a short time (compared to manual testing) and potential defects can be found quicker. In my opinion, automated tests are more interesting, because you can focus your attention on creating and upgrading already existing test scenarios (e.g. by randomizing test data or adding more complex use cases), instead of on performing the same test manually over and over again. One more thing worth mentioning is that tests' results can be seen by every team member involved in the project, as you can automatically create test logs - for example, you can include what each suite is testing, what test data was used or what type of error occurred and failed the test.

Each Python test file that runs on unittest module, consists of several important parts that need to be in place in order to successfully run the test. At the very top, there are a couple import statements - those are always set automatically when you export tests from Selenium. Not all of them are used in our example, but there's no harm in leaving it as it is.



```

from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import Select
from selenium.common.exceptions import NoSuchElementException
from selenium.common.exceptions import NoAlertPresentException
import unittest, time, re
```

Basically, all commands have high-level names and are quite self-explanatory. You should notice that they have a specific structure - first, there is a webdriver variable, then there's an element you want to interact with (notice the different types of selectors) and then you have a specified action you want to perform on this element.