Chapter 6

Testing

A lot of programmers agree that “testing” is one of the most important aspects of the development of the software. Testing is the art of generating codes that can test our programs, checking that our development achieves the behavior requested by the final users. Even though we generally perform manual tests each time we develop a new piece of code that executes a task, it’s very likely that we manually test for a rather typical case, which does not ensure that our new piece of code works in every possible scenario. Another factor to consider is efficiency: it takes a lot of time and code lines to set up and execute each evaluation. For these reasons, programmers prefer to automate testing since it quickly creates the set up for many different tests and allows to run them in a much visible way.

From now on, all your programs must go hand in hand with a program that tests it. On this chapter, we will see the fundamental concepts of testing and how to assemble unitary tests, but testing is a section that provides for an entire course.

In this chapter, we focus on unitary tests. A unitary test performs tests on minimal units of codes, such as functions or class methods. We present two Python libraries that make easier the creation of unitary tests: *unittest* y *Pytest*.

6.1 Unittest

The library `unittest` of Python gives many tools to create and run tests, one of the most important classes is “TestCase”. In general, the classes that we create to perform tests must inherit the `TestCase` class. By convention all of the methods we implement to test must be called starting with the word “test”, so that they are recognize at the moment of running the full program of testing in an automatic way:

```python
import unittest

```

```python

class CheckNumbers(unittest.TestCase):

    # This test has to be ok
    def test_int_float(self):
        self.assertEquals(1, 1.0)

    # This test fails
    def test_str_float(self):
        self.assertEquals(1, "1")

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

```
.F
======================================================================
FAIL: test_str_float (__main__.CheckNumbers)
----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

The dot before the F indicates that the first test (test_int_float) passed with success. The F after the dot indicates that the second test failed. Afterward appear the details of the tests that failed, followed by the number of executed tests, the time it took and the total number of tests that failed.

We could then have all the tests we need inside the class that inherits TestCase, as long as the name of the method begins with "test". Each test must be entirely independent of the others, in other words, the result of the calculus of a test must not affect the result of another test.

Assertion Methods

Assertion methods allow us to perform an essential kind of tests, where we know the desired result, and we just check if the value returned by the test matches that result. Assertion methods allow us to validate results in different ways. Some assertion methods (included in the class TestCase) are:

```python
import unittest


class ShowAsserts(unittest.TestCase):

    def test_assertions(self):
        a = 2
        b = a
        c = 1. + 1.
        self.assertEqual([1, 2, 3], [1, 2, 3])  # Fails if a != b
        self.assertNotEqual("hello", "bye")  # Fails if a == b
        self.assertTrue("Hello" == "Hello")  # Fails if bool(x) is False
        self.assertFalse("Hello" == "Bye")  # Fails if bool(x) is True
        self.assertIs(a, b)  # Fails if a is not b

        # Fails if a is b. Pay attention that "is" implies
        # equality (==) but not upside. Two differents objects
        # can have the same value.
        self.assertIsNot(a, c)

        self.assertIsNone(None)  # Fails if x is not None
        self.assertIsNotNone(2)  # Fails if x is None
        self.assertIn(2, [2, 3, 4])  # Fails if a is not in b
        self.assertNotIn(1, [2, 3, 4])  # Fails if a is in b

        # Fails if isinstance(a, b) is False
        self.assertIsInstance("Hello", str)
        # Fail if isinstance(a, b) is True
        self.assertNotIsInstance("1", int)

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

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

```
OK
```

The method `assertRaises` requires an exception, a function or any object with the implemented method `__call__` (*callable*) and an arbitrary number of arguments that will be passed to the *callable* method. The assertion will invoke the *callable* method with its arguments. It will fail if the method does not generate the expected error. The next code shows two ways of how to use the method `assertRaises`.

```python
import unittest


def average(seq):
    return sum(seq) / len(seq)


class TestAverage(unittest.TestCase):

    def test_python30_zero(self):
        self.assertRaises(ZeroDivisionError, average, [])

    def test_python31_zero(self):
        with self.assertRaises(ZeroDivisionError):
            average([])

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

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

In the second example `assertRaises` is used within a context manager (`with` sentence). It allows us to write our code in a more natural way by calling directly to the function `average` instead of having to call it indirectly as in the other example. We address context managers with more details in Chapter 10.

In Python 3.4 appeared new assertion methods:

- `assertGreater(first, second, msg=None)`
- `assertGreaterEqual(first, second, msg=None)`
- `assertLess(first, second, msg=None)`
- `assertLessEqual(first, second, msg=None)`
- `assertAlmostEqual(first, second, places=7, msg=None, delta=None)`
- `assertNotAlmostEqual(first, second, places=7, msg=None, delta=None)`

They test that `first` and `second` are approximately (or not approximately) equal, by calculating the difference, rounding the result number to `places` decimals places. If the argument `delta` is provided instead of “places”, the difference between first and second must be less or equal (o more in case of `assertNotAlmostEqual`) than delta. If `delta` and `places` are given it generates an error.

The setUp method

Once we have written several tests, we realize we need to assemble a group of objects that will be used as input to compare the results of a test. The `setUp` method allows us to declare the variables that we will use and also takes care of resetting or re-initializing the variables before entering to a new test, in case that one of the other tests modified something in the variables.

```python
from collections import defaultdict
import unittest


class StatisticList(list):

    def mean(self):
        return sum(self) / len(self)

    def median(self):
        if len(self) % 2:
            return self[int(len(self) / 2)]
        else:
            idx = int(len(self) / 2)
            return (self[idx] + self[idx-1]) / 2

    def mode(self):
        freqs = defaultdict(int)
        for item in self:
            freqs[item] += 1
        mode_freq = max(freqs.values())
        modes = []
        for item, value in freqs.items():
            if value == mode_freq:
                modes.append(item)
        return modes


class TestStatistics(unittest.TestCase):

    def setUp(self):
        self.stats = StatisticList([1, 2, 2, 3, 3, 4])

    def test_mean(self):
        print(self.stats)
        self.assertEqual(self.stats.mean(), 2.5)

    def test_median(self):
        self.assertEqual(self.stats.median(), 2.5)
        self.stats.append(4)
        self.assertEqual(self.stats.median(), 3)

    def test_mode(self):
        print(self.stats)
        self.assertEqual(self.stats.mode(), [2, 3])
        self.stats.remove(2)
        self.assertEqual(self.stats.mode(), [3])
```

```

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

```
[1, 2, 2, 3, 3, 4]
..[1, 2, 2, 3, 3, 4]
.
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Notice that the `setUp` method is never called explicitly inside of any of the tests because `unittest` does that for us. As you can see `test_median` modifies the list by adding a 4, but then in `test_mode` the list is the same as in the beginning. It occurs because `setUp` manages to re-initialize the variables that we need at the start of the each test. It helps us not to repeat code needlessly.

Besides the method `setUp`, `TestCase` offers us the method `tearDown`, which can be used to "clean" after all the tests have been executed. For instance, if our tests have the need to create some files, the idea is that at the end all of those temporary files are eliminated, in a way that ensures that the system is in the same state that it was before executing the tests:

```python
import os
import unittest


class TestFiles(unittest.TestCase):

    def setUp(self):
        self.file = open("test_file.txt", 'w')
        self.dictionary = {1: "Hello", 2: "Bye"}

    def tearDown(self):
        self.file.close()
        print("Removing temporary files...")
        os.remove("test_file.txt")

    def test_str(self):
        print("Writing temporary files...")
        self.file.write(self.dictionary[1])
        self.file.close()
        self.file = open("test_file.txt", 'r')
        d = self.file.readlines()[0]
        print(d)
        self.assertEqual(self.dictionary[1], d)


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

```
Writing temporary files...
Hello
Removing temporary files...
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

The discover module

When we test a program, we quickly begin to fill us with testing codes only. To solve this problem, we can arrange our modules that contain tests (objects TestCase) in more general modules called test suites (objects TestSuite), which include collections of tests:

```python
import unittest


class ArithmeticTest(unittest.TestCase):

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

```

```

if __name__ == "__main__":
    Tsuite = unittest.TestSuite()
    Tsuite.addTest(unittest.TestLoader().loadTestsFromTestCase(ArithmeticTest))
    unittest.TextTestRunner().run(Tsuite)
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

How to ignore tests

Many times we know that some of the tests are going to fail in our program and we do not want they to fail during a given test. For example, when we have a function that is not yet finished. In these particular cases, we would like that the suite does not run these tests because we already know they will fail. Fortunately, *unittest* provides us with some decorators that mark tests and ignore them under certain circumstances. These decorators are: `expectedFailure()`, `skip(reason)`, `skipIf(condition, reason)`, `skipUnless(condition, reason)`:

```
import unittest
import sys


class IgnoreTests(unittest.TestCase):

    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(False, True)

    @unittest.skip("Useless test")
    def test_ignore(self):
        self.assertEqual(False, True)

    @unittest.skipIf(sys.version_info.minor == 5, "does not work on 3.5")
```

```python
    def test_ignore_if(self):
        self.assertEqual(False, True)

    @unittest.skipUnless(sys.platform.startswith("linux"),
                         "does not work on linux")
    def test_ignore_unless(self):
        self.assertEqual(False, True)


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

```
xsFs
======================================================================
FAIL: test_ignore_if (__main__.IgnoreTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "src/ENG/chapter_06/codes/unittest/6_ignore_test.py", line 17,
  in test_ignore_if self.assertEqual(False, True)
AssertionError: False != True

----------------------------------------------------------------------
Ran 4 tests in 0.001s
```

The `x` in the first line means *expected failure*, the `s` means *skipped test*, and the `F` means a real failure.

6.2 Pytest

Pytest is a framework for an alternative testing of *unittest*. It has a different design and allows to write the test in a more simple and readable way. *Pytest* also does not require that testing cases are classes, taking advantage of the fact that functions are objects, allowing any suitable function to work as a test. These features make the tests written in *Pytest* more readable and maintainable.

When we execute `py.test`, it begins by searching all the modules or sub-packages that begin with `test` in the current folder. Any function in that module that also begins with `test_` will be executed as an individual test. Whenever exists some class (inside the module) that starts with `Test`, any method inside that class beginning with `test_` will also be executed in the testing environment. The next example shows how to write simple unitary tests; we can see that it is much shorter than it *unittest* counterparts:

```python
def test_int_float():
    assert 1 == 1.0


def test_int_str():
    assert 1 == "1"
```

If we execute this code with `py.test PyTest0.py` in console the output is (assuming that in the file *PyTest0.py* is the code with the tests):

```
============================= test session starts ======================
platform win32 -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: codes\pytest, inifile:
collected 2 items

PyTest0.py .F

================================== FAILURES ============================
________________________________ test_int_str __________________________

    def test_int_str():
>       assert 1 == "1"
E       assert 1 == '1'

PyTest0.py:7: AssertionError
===================== 1 failed, 1 passed in 0.02 seconds ===============
```

We can see in the printed output some information about the platform. After that appears the name of the file that contains the tests; then we see the same notation used in *unittest* about the tests results, in this case anew the dot (".") indicates that the test has passed and the `F` that the test failed. We also see that it highlights the error and it mentions which kind of error it is.

We can also use classes in *pytest*. In this case, we do not have to inherit from any superclass from the testing module, as we did in *unittest*):

```
class TestNumbers:

    def test_int_float(self):
        assert 1 == 1.0

    def test_int_str(self):
        assert 1 == "1"
```

```
============================= test session starts =======================
platform win32 -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: codes\pytest, inifile:
collected 2 items

PyTest1.py .F

================================== FAILURES ==============================
__________________________ TestNumbers.test_int_str _____________________

self = <PyTest1.TestNumbers object at 0x000001661A37EBE0>

    def test_int_str(self):
>       assert 1 == "1"
E       assert 1 == '1'

PyTest1.py:7: AssertionError
===================== 1 failed, 1 passed in 0.03 seconds ================
```

`setup` and `teardown` in Pytest

The main difference between the `setup` and `teardown` methods from *pytest* (more precisely `setup_method` and `teardown_method`) with the ones from *unittest*, is that in *pytest* these methods accept one argument: the function (object) that represents the method that is being called.

In addition, *pytest* provides the `setup_class` and `teardown_class` functions, that are methods that must test one class, they actually receive the class they are going to test as an argument

Finally, the `setup_module` and `teardown_module` methods are functions that run before and after all tests (despite they are functions or classes) in the module. The next example shows how to use each of these methods and in which order they are being executed:

```python
def setup_module(module):
    print("Setting up module {0}".format(module.__name__))


def teardown_module(module):
    print("Tearing down module {0}".format(module.__name__))


def test_a_function():
    print("Running test function")


class BaseTest:

    def setup_class(cls):
        print("Setting up Class {0}".format(cls.__name__))

    def teardown_class(cls):
        print("Tearing down Class {0}\n".format(cls.__name__))

    def setup_method(self, method):
        print("Setting up method {0}".format(method.__name__))

    def teardown_method(self, method):
        print("Tearing down  method {0}".format(method.__name__))


class TestClass1(BaseTest):

    def test_method_1(self):
        print("Running Method 1-1")

    def test_method_2(self):
        print("Running Method 2-1")


class TestClass2(BaseTest):

    def test_method_1(self):
        print("Running Method 1-2")

    def test_method_2(self):
        print("Running Method 2-2")

# Running by console "py.test PyTest.py -s",
# -s disables the deletion of "print" outputs.
```

```
============================= test session starts ======================
platform win32 -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: \codes\pytest, inifile:
collected 5 items

PyTest2.py Setting up module PyTest2
Running test function
.Setting up Class TestClass1
Setting up method test_method_1
Running Method 1-1
.Tearing down  method test_method_1
Setting up method test_method_2
Running Method 2-1
.Tearing down  method test_method_2
Tearing down Class TestClass1

Setting up Class TestClass2
Setting up method test_method_1
Running Method 1-2
.Tearing down  method test_method_1
Setting up method test_method_2
```

```
Running Method 2-2
.Tearing down  method test_method_2
Tearing down Class TestClass2

Tearing down module PyTest2


========================== 5 passed in 0.02 seconds ====================
```

Funcargs in pytest

A different way to do a setup and teardown in *pytest* is using *funcargs* (an abbreviation for function arguments). They correspond to variables that are previously set up in a file of tests configuration. It allows us to separate the configuration of the execution files from the tests files, and it also makes it possible to use the *funcargs* through multiple classes and modules.

To use the *funcargs*, we just add parameters to our testing functions, the names of these parameters will be used to search specific arguments in functions with a particular name (with the example this will become clearer). For instance, if we want to use the class `StatisticsList` in one of the previous examples by using *funcargs*:

```python
def pytest_funcarg__valid_stats(request):
    return StatisticsList([1, 2, 2, 3, 3, 4])


def test_mean(valid_stats):
    assert valid_stats.mean() == 2.5


def test_median(valid_stats):
    assert valid_stats.median() == 2.5
    valid_stats.append(4)
    assert valid_stats.median() == 3


def test_mode(valid_stats):
    assert valid_stats.mode() == [2, 3]
    valid_stats.remove(2)
    assert valid_stats.mode() == [3]
```

```
============================ test session starts =====================
platform win32 -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: codes\pytest, inifile:
collected 3 items

PyTest3.py EEE

=================================== ERRORS ============================
_________________________ ERROR at setup of test_mean _________________

request = <SubRequest 'valid_stats' for <Function 'test_mean'>>

    def pytest_funcarg__valid_stats(request):
>       return StatisticsList([1, 2, 2, 3, 3, 4])
E       NameError: name 'StatisticsList' is not defined

PyTest3.py:3: NameError
________________________ ERROR at setup of test_median ________________

request = <SubRequest 'valid_stats' for <Function 'test_median'>>

    def pytest_funcarg__valid_stats(request):
>       return StatisticsList([1, 2, 2, 3, 3, 4])
E       NameError: name 'StatisticsList' is not defined

PyTest3.py:3: NameError
_________________________ ERROR at setup of test_mode _________________

request = <SubRequest 'valid_stats' for <Function 'test_mode'>>

    def pytest_funcarg__valid_stats(request):
>       return StatisticsList([1, 2, 2, 3, 3, 4])
E       NameError: name 'StatisticsList' is not defined
```

```
PyTest3.py:3: NameError
=========================== 3 error in 0.05 seconds ===================
```

Note that the function has the prefix `pytest_funcarg__`. `valid_stats` contains the parameters used by the testing methods. In this case, `valid_stats` is returned by the `pytest_funcarg__valid_stats` function and corresponds to the list `StatisticsList([1,2,2,3,3,4])`. Note that in general the *funcargs* should be defined in an apart file called conftest.py, to allow the function be used by several modules. From the example we can see that the *funcargs* are reloaded every time for each test, hence we can modify the list inside a test as much as we need, without affecting other tests that may use the *funcargs* in further executions.

The `request.addfinalizer` method receives a function for cleaning purposes. It is equivalent to the `teardown` method, for example, it allows us to clean files, close connections, and empty lists. The next example tests the `os.mkdir` method (creates a directory) by checking that two temporary directories are indeed created:

```python
import tempfile
import shutil
import os.path


def pytest_funcarg__temp_dir(request):
    dir = tempfile.mkdtemp()
    print(dir)

    def cleanup():
        shutil.rmtree(dir)

    request.addfinalizer(cleanup)
    return dir


def test_osfiles(temp_dir):
    os.mkdir(os.path.join(temp_dir, 'a'))
    os.mkdir(os.path.join(temp_dir, 'b'))
    dir_contents = os.listdir(temp_dir)
    assert len(dir_contents) == 2
    assert 'a' in dir_contents
    assert 'b' in dir_contents
```

```
============================= test session starts ====================
platform win32 -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: codes\pytest, inifile:
collected 1 items

PyTest4.py .

========================== 1 passed in 0.02 seconds ==================
```

Skip tests in pytest

As in *unittest*, in *pytest* we can ignore some tests by using the `py.test.skip` function. It accepts only one argument; a string describing why we are ignoring the test. We can call `py.test.skip` anywhere. If we call it inside a test function, the test will be ignored; if we call it inside a module, all the tests within the module will be ignored, and if we call it inside a *funcarg* function, all the tests that call that *funcarg* will be ignored:

```python
import sys
import py.test


def test_simple_skip():
    if sys.platform != "Linows":
        py.test.skip("This test only works on Linows OS")
    fakeos.do_something_fake()
    assert fakeos.did_not_happen
```

```
============================= test session starts ====================
platform win32 -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: codes\pytest, inifile:
collected 1 items

PyTest5.py s

========================== 1 skipped in 0.02 seconds =================
```

Note that we can call the function that ignores the tests inside of `if` sentences, which gives us a lot of conditioning possibilities. Besides the `skip` function, *pytest* offers us a decorator that allows us to skip tests when a particular condition is fulfilled. The decorator is `@py.test.mark.skipif(string)`, where the argument it receives corresponds to a string that contains a code that returns a boolean value after its execution.

```python
import sys
import py.test


@py.test.mark.skipif("sys.platform != \"Linows\"")
def test_simple_skip():
    fakeos.do_something_fake()
    assert fakeos.did_not_happen
```

```
============================= test session starts =====================
platform win32 -- Python 3.5.2, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: codes\pytest, inifile:
collected 1 items

PyTest6.py s

========================== 1 skipped in 0.01 seconds =================
```

In *pytest*, there is another way to obtain a setup/teardown framework. This way is called *fixtures*. We will not address *fixtures* because we believe that it is out of the scope of this book, but you can find details about *fixtures* in this link: `https://pytest.org/latest/fixture.html\#fixture`.

6.3 Hands-On Activities

Activity 6.1

A management system has lately suffered attacks to its servers. The managers decided to increase their security by adding encryption to the information. You must test the encryption system.

The only things you know about the system are the following:

- The encrypter is symmetrical, which means, to encrypt and decrypt information, we use the same function. This is described in the `Encoder` class. The method to encrypt is called `encrypt`.

- The encrypter is based in two classes, called `Rotor` and `Reflector`.
- An *alphabet* is a set of allowed characters, it must be added before the encryption begins. To test the program, you can assume that the alphabet is lowercase ASCII. You must perform a test setup by calling the method `create_alphabet`, as shown in `main`.
- If we enter a word with characters that are not included in the alphabet, the program will raise a `ValueError` exception.

Rotor method

It takes a text archive and creates a function from it. It is not necessary that you know how it works, you only have to assert the properties of the function.

The application of this function is done through the `get` method. If the values are not in the domain, it returns `None`. To work correctly, it is necessary that this function is bijective.

Reflector method

It takes a text archive. The reflector is equal to the rotor, but it has a symmetry function. If $f(x) = y$ then $f(y) = x$. You access the function in the class through the method `get`. If the value is not in the domain, it returns `None`. It must show that bijection and symmetry hold. To test symmetry is enough to see if `get(x)`=y with `y` $\neq$ `None`. Also `get(y)=x` must hold.

Encrypter method

It has as initialization parameters a list of the rotors' addresses and the reflector's address.

To test that the encryption is correct you must prove for the following list of strings that by calling `encrypt`, the word gets encrypted and by calling it again, the word becomes decrypted.

`list_ = ['thequickbrownfoxjumpsoverthelazydog', 'python', 'bang', 'dragonfly', 'csharp'].`

To test the exception is enough to try with any word that has characters that are not in the alphabet. For instance, you may attempt with `'ñandu'`.

Notes

- Your code can be based in `encoder_test.py`.
- Check the `main` method if you have doubts on how to use the encrypter.
- To test the bijection of the functions, you can call the function `check_bijection` of the base code that receives a rotor or a reflector, and prove it holds bijection.

Summarizing, you have to perform the following steps:

- Initialize the alphabet as ASCII lowercase with a setup module.
- Prove that the rotor function is bijective. You must prove this for the archives (`rotor1.txt, rotor2.txt, rotor3.txt`)
- Prove that the reflector function is bijective and symmetric. You must prove this for the archive (`reflector.txt`)
- Prove that the encryption and decryption is correct. This includes to test that it raises the exception when needed.

Activity 6.2

The Bank has recently acquired last technology ATMs. The ATMs need to be programmed, for this reason, the bank has contacted you. You must assure that the coded features work correctly.

The features that come already implemented and you must test are:

- **Withdraw cash**: The user chooses an amount and withdraws it from its account. To obtain the money, the user must enter his password correctly. The maximum amount per transaction is $200.
- **Transfer money to a third party**: The user can transfer money to other users. The maximum amount to be transferred per operation is $1000.

For the withdraw money feature, create the following tests with **unittest**:

- Check that the credentials are correct.
- Check that it only withdraws money if there is enough balance.

- Check that once the money has been withdrawn, the amount is updated.

For the transfer money feature, create the following tests with **unittest**:

- Check that the account of the third person exists.
- Check that once the money has been transferred, both accounts have been updated with the correct amount.
- Check that if there is any error, the transference did not occur.