Claus Führer, Jan Erik Solem and Olivier Verdier

Computing with Python

An Introduction to Python for Science and Engineering
Chapter XIV

Testing

In this chapter we focus on two aspects of testing for scientific programming. The first aspect is the often difficult topic of what to test in scientific computing. The second aspect covers the question how to test. We will distinguish between manual and automated testing. Manual testing is what is done by every programmer to quickly check that an implementation is working. Automated testing is the refined, automated variant of that idea. We will introduce some tools available for automatic testing in general, with a view on the particular case of scientific computing.

1 Manual Testing

During the development of code you do a lot of small tests in order to test its functionality. This could be called manual testing. Typically, you would test that a given function does what it is supposed to do, by manually testing the function in an interactive environment.

For instance, suppose that you implement the bisection algorithm. It is an algorithm that finds a zero (root) of a scalar nonlinear function. To start the algorithm an interval has to be given with the property, that the function takes different signs on the interval boundaries, see Exercise VIII.4.

You would then test an implementation of that algorithm typically by checking

1. that a solution is found when the function has opposite signs at the interval boundaries
2. that an exception is raised when the function has the same sign at the interval boundaries

Manual testing, as necessary as may seem to be, is unsatisfactory:
Once you convinced yourself that the code does what it is supposed to do, you formulate a relatively small number of demonstration examples to convince others of the quality of the code. At that stage one often looses interest in the tests made during development and they are forgotten or even deleted.
As soon as you change a detail and things no longer work correctly you might regret that your earlier tests are no longer available.

2 Automatic Testing

The correct way to develop any piece of code is to use automatic testing. The advantages are

- the automated repetition of a large number of tests after every code refactoring\(^1\) and before new versions are launched
- a silent documentation of the use of the code
- a documentation of the test coverage of your code: Did things work before a change or was a certain aspect never tested?

We suggest to develop tests in parallel to the code. Good design of tests is an art of its own and there is rarely an investment which guarantees such a good pay-off in development time savings as the investment in good tests.
Now we will go though the implementation of a simple algorithm with the automated testing methods in mind.

2.1 Testing the bisection algorithm

Let us examine automated testing for the bisection algorithm. With this algorithm a zero of a real valued function is found. It is described in Exercise VIII.4.

An implementation of the algorithm can have the following form:

```python
def bisect(f, a, b, tol=1.e-8):
    """
    Implementation of the bisection algorithm
    """
```

\(^1\)Changes in the program and in particular in its structure which do not affect its functionality are called code refactoring.
2. Automatic Testing

```python
f real valued function
a,b interval boundaries (float) with
    the property f(a) * f(b) <= 0
tol tolerance (float)
    if f(a) * f(b) >0:
        raise ValueError("Incorrect initial interval [a, b]")
    for i in xrange(100):
        c = (a + b) / 2.
        if f(a) * f(c) <= 0:
            b = c
        else:
            a = c
        if abs(a - b) < tol:
            return (a + b) / 2
        raise Exception('No root found within the given tolerance {}
                          .format(tol)
```

We assume this to be stored in a file `bisection.py`.

As a first test case we test that the zero of the function \( f(x) = x \) is found:

```python
def test_identity():
    result = bisect(lambda x: x, -1., 1.) # (for lambda see § VIII.7.)
    expected = 0.
    assert allclose(result, expected),'expected zero not found'
text_identity()
```

In this code you meet the Python keyword `assert` for the first time. It raises an exception `AssertionError` if its first argument returns the value `False`. Its optional second argument is a string with additional information.

We use the function `allclose` in order to test for equality for floats, see § III.2.3.

Let us comment on some of the features of the test function. We use an `assertion` to make sure that an exception will be raised if the code does not behave as expected.

We have to manually run the test in the line `test_identity()`. There are many tools to automate this kind of call, we will see one of those in § 2.2.

Let us now setup a test that checks if `bisect` raises an exception when the function has the same sign on both ends of the interval. For now, we will suppose that the exception raised is a `ValueError` exception.
Example 1.

Checking the sign for the bisection algorithm.

```python
def test_badinput():
    try:
        bisect(lambda x: x, 0.5, 1)
    except ValueError:
        pass
    else:
        raise AssertionError()

test_badinput()
```

In this case an `AssertionError` is raised if the exception is not of type `ValueError`.

There are tools to simplify the above construction to check that an exception is raised.

Another useful kind of tests is the edge case test. Here we test arguments or user input which is likely to create mathematically undefined situations or states of the program not foreseen by the programmer.

For instance, what happens if both bounds are equal? What happens if $a > b$? We easily setup up such a test by using for instance

```python
def test_equal_boundaries():
    result = bisect(lambda x: x, 1., 1.)
    expected = 0.
    assert allclose(result, expected), 'test equal interval bounds failed'
def test_reverse_boundaries():
    result = bisect(lambda x: x, 1., -1.)
    expected = 0.
    assert allclose(result, expected), 'test reverse interval bounds failed'

test_equal_boundaries()
test_reverse_boundaries()
```

2.2 Using unittest

The standard Python package `unittest` greatly facilitates automated testing. That package requires that we rewrite our tests a little to be compatible.

The first test would have to be rewritten in a `class`, as follows:
from bisection import bisect
import unittest

class TestIdentity(unittest.TestCase):
    def test(self):
        result = bisect(lambda x: x, -1.2, 1., tol=1.e-8)
        expected = 0.
        self.assertAlmostEqual(result, expected)

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

Let us examine the differences to the previous implementation. First, the test is now a method and a part of a class. The class must inherit from unittest.TestCase. The test method’s name must start with test. Note that we may now use one of the assertion tools of the unittest package, namely assertAlmostEqual. Finally, the tests are run using unittest.main.

We recommend to write the tests in a file separate from the code to be tested. That’s why it starts with an import.

The test passes and returns

Ran 1 test in 0.002s

OK

If we would have run it with a too loose tolerance parameter, e.g., 1.e-3, a failure of the test would have been reported:

F

======================================================================
FAIL: test (__main__.TestIdentity)
--------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-11-e44778304d6f>", line 5, in test
    self.assertAlmostEqual(result, expected)
AssertionError: 0.00017089843750002018 != 0.0 within 7 places

Ran 1 test in 0.004s

FAILED (failures=1)

Tests can and should be grouped together as methods of a test class:

Example 2.

import unittest

```python
```
class TestIdentity(unittest.TestCase):
    def identity_fcn(self, x):
        return x
    def test_functionality(self):
        result = bisect(self.identity_fcn, -1.2, 1., tol=1.e-8)
        expected = 0.
        self.assertAlmostEqual(result, expected)
    def test_reverse_boundaries(self):
        result = bisect(self.identity_fcn, 1., -1.)
        expected = 0.
        self.assertAlmostEqual(result, expected)
    def test_exceeded_tolerance(self):
        tol = 1.e-80
        self.assertRaises(Exception, bisect, self.identity_fcn, -1.2, 1., tol)

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

Here, the last test needs some comments: We used the method unittest.TestCase.assertRaises. It tests whether an exception is correctly raised. Its first parameter is the exception type, e.g., ValueError, Exception, and its second argument is a the name of the function, which is expected to raise the exception. The remaining arguments are the arguments for this function.

The command unittest.main() creates an instance of the class TestIdentity and executes those methods starting by test.

### 2.3 Test setUp and tearDown

The class unittest.TestCase provides two special methods, setUp and tearDown, which are run before and after every call to a test method. This is needed when testing generators, which are exhausted after every test.

We demonstrate this here by testing a program which checks in which line in a file a given string occurs for the first time:

class NotFoundError(Exception):
    pass

def find_string(file, string):
    for i, lines in enumerate(file.readlines()):
        if string in lines:
            return i
2. Automatic Testing

We assume, that this code is saved in a file `find_string.py`.
A test has to prepare a file and open it and remove it after the test:

```python
import unittest
import os  # used for, e.g., deleting files
from find_in_file import find_string, NotFoundError

class TestFindInFile(unittest.TestCase):
    def setUp(self):
        file = open('test_file.txt', 'w')
        file.write('aha')
        file.close()
        self.file = open('test_file.txt', 'r')
    def tearDown(self):
        os.remove(self.file.name)
    def test_exists(self):
        line_no = find_string(self.file, 'aha')
        self.assertEqual(line_no, 0)
    def test_not_exists(self):
        self.assertRaises(NotFoundError, find_string, self.file, 'bha')

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

Before each test `setUp` is run and afterwards `tearDown` is executed.

2.4 Parametrizing Tests

One frequently wants to repeat the same test set-up with different data sets. When using the functionalities of unittests this requires to automatically generate test cases with the corresponding methods 'injected':

To this end we first construct a test case with one or several methods that will be used, when we later set up test methods. Let us consider the bisection method again and let us check if the values it returns are really zeros of the given function.

We first build the test case and the method which will use for the tests:

```python
class Tests(unittest.TestCase):
    def checkifzero(self, fcn_with_zero, interval):
        result = bisect(fcn_with_zero, *interval, tol=1.e-8)
        function_value = fcn_with_zero(result)
        expected = 0.
        self.assertAlmostEqual(function_value, expected)
```
Then we dynamically create test functions as attributes of this class:

```python
test_data=[{'name':'identity', 'function':lambda x: x, 'interval':[-1.2, 1.1]},
           {'name':'parabola', 'function':lambda x: x**2-1, 'interval':[0, 10.1]},
           {'name':'cubic', 'function':lambda x: x**3-2*x**2, 'interval':[0.1, 5.1]}]

def make_test_function(dic):
    return lambda self:self.checkifzero(dic['function'],dic['interval'])

for data in test_data:
    setattr(Tests, "test_{name}".format(name=data['name']),
            make_test_function(data))
```

In this example the data is provided as a list of dictionaries. A function `make_test_function` dynamically generates a test function which uses a particular data dictionary to perform the test with the previously defined method `checkifzero`. This test function is made a method of the `TestCase` class by using the Python command `setattr`.

2.5 Assertion Tools

In this section we collect the most important tools for raising an `AssertionError`. We met already the command `assert` and two tools from `unittest`, namely `assertEquals`.

Table Table XIV.1 summarizes the most important assertion tools and the related modules.

2.6 Float Comparisons

Two floating point numbers should not be compared with the `==` comparison, because the result of a computation is often slightly off, due to rounding errors. There are numerous tools to test equality of floats for testing purposes.

First, as mentioned in § III.2.3, `allclose` checks that two arrays are almost equal. It can be used in a test function like

```python
self.assertTrue(allclose(computed , expected))
```

Here, `self` refers to a `unittest.TestCase` instance.
2. Automatic Testing

There are also testing tools in the numpy package `testing`. These are imported by

```python
import numpy.testing
```

Testing that two scalars or two arrays are equal is done using `numpy.testing.assert_array_almost_equal` or `numpy.testing.assert_allclose`. These methods differ in the way they describe the required accuracy, see Table XIV.1.

Example 3.

$QR$ factorization decomposes a given matrix into a product of an orthogonal matrix $Q$ and an upper triangular matrix $R$:

```python
import scipy.linalg as sl
A=rand(10,10)
[Q,R]=sl.qr(A)
```

Is the method applied correctly? We can check this by verifying that $Q$ is indeed an orthogonal matrix:

```python
import numpy.testing as npt
npt.assert_allclose(dot(self.Q.T,self.Q),identity(self.Q.shape[0]),atol=1.e-12)
```

Furthermore we might perform a sanity test by checking if $A = QR$:

```python
import numpy.testing as npt
npt.assert_allclose(dot(Q,R),A)
```

**assertion tool and application example** | **module**
---|---
assert $5==5$ | _
assertEqual (5.27, 5.27) | unittest.TestCase
assertAlmostEqual (5.24, 5.2,places = 1) | unittest.TestCase
assertTrue (5 > 2) | unittest.TestCase
assertFalse (2 < 5) | unittest.TestCase
assertRaises (ZeroDivisionError,lambda x: 1/x,0.) | unittest.TestCase
assertIn (3, {3, 4}) | unittest.TestCase
assert_array_equal (A,B) | numpy.testing
assert_array_almost_equal (A, B, decimal=5) | numpy.testing
assert_allclose (A, B, rtol=1.e-3, atol=1.e-5) | numpy.testing

Table XIV.1: Most common assertion tools
All this can be collected into a unittest testcase:

```python
import unittest
import numpy.testing as npt
from scipy.linalg import qr
from scipy import *

class TestQR(unittest.TestCase):
    def setUp(self):
        self.A=rand(10,10)
    def test_orthogonal(self):
        npt.assert_allclose(dot(self.Q.T,self.Q),identity(self.Q.shape[0]),atol=1.e-12)
    def test_sanity(self):
        npt.assert_allclose(dot(self.Q,self.R),self.A)
```

Note in assert_allclose the parameter atol defaults to 0, which often causes problems, when working with matrices having small elements.

2.7 Unit and Functional Tests

Up to now, we have only used functional tests. A functional test checks that the functionality is correct. For the bisect algorithm, it is indeed, that the algorithm actually finds a zero, when there is one.

In that simple example it is not really clear what a unit test is. Although it might seem slightly contrived, it is still possible to make a unit test for the bisection algorithm. It will demonstrate how unit testing often leads to more compartimented implementation.

So, in the bisection method, we would like to check, for instance, that at each step, the interval is chosen correctly. How to do that? Note that it is absolutely impossible with the current implementation, because the algorithm is hidden inside the function.

One possible remedy is to run only one step of the bisection algorithm. Since all the steps are similar, we might argue that we have tested all the possible steps. We also need to be able to inspect the current bounds \( a \) and \( b \) at the current step of the algorithm.

So we have to add the number of steps to be run as a parameter, and change the return interface of the function. We do that as follows:

```python
def bisect(f,a,b,n=100):
    ...
    for iteration in xrange(n):
```
2. Automatic Testing

```
... return a, b
```

Note that we have to change the existing unit tests in order to accommodate for that change.

We may now add a unit test as follows

```python
def test_midpoint(self):
a, b = bisect(identity, -2., 1., 1)
self.assertAlmostEqual(a, -0.5)
selk.assertAlmostEqual(b, 1.)
```

2.8 Debugging

Debugging is sometimes necessary while testing, in particular if it is not immediately clear why a given test does not pass. In that case it is useful to be able to debug a given test in an interactive session. This is however made difficult by the design of the `unittest.TestCase` class, which prevents easy instantiation of test case objects.

The solution is to create a special instance for debugging purpose only. Suppose that, in Example 2, we want to test the method `test_functionality`. This would be achieved by

```python
test_case = TestIdentity(methodName='test_functionality')
```

Now this test can be run individually by

```python
test_case.debug()
```

This will run this individual test and allows for debugging.

2.9 Test Discovery

If you write a Python package various tests might be spread out through the package. The `discover` module finds, imports and runs these test cases. The basic call from the command line is

```bash
python -m unittest discover
```

It starts looking for test cases in the current directory and recurses the directory tree downward to find Python objects with the string `test` contained in its name.

The command takes optional arguments. Most important are `-s` to modify the start directory and `-p` to define the pattern to recognize tests:

```bash
python -m unittest discover -s '.' -p 'test*.py'
```
XIV. Testing

3 Measuring Execution Time

In order to take decisions on code optimization, one often has to compare several code alternatives and decide which code should be preferred based on the execution time.

Furthermore, discussing execution time is an issue when comparing different algorithms. In this section we present a simple an easy way to measure execution time.

3.1 Timing with a magic function

The easiest way to measure the execution time of a single statement is to use IPython's magic function `%timeit`\(^2\). As the execution time of a single statement can be extremely short, the statement is placed in a loop and executed several times. By taking the minimum measured time, one makes sure that other tasks running on the computer do not influence the measured result too much.

Example 4.

Let us consider four alternative ways to extract nonzero elements from an array:

```python
A=zeros((1000,1000))
A[53,67]=10

def find_elements_1(A):
    b = []
    n, m = A.shape
    for i in xrange(n):
        for j in xrange(m):
            if abs(A[i, j]) > 1.e-10:
                b.append(A[i, j])
    return b

def find_elements_2(A):
    return [a for a in A.reshape((-1, )) if abs(a) > 1.e-10]

def find_elements_3(A):
    return [a for a in A.flatten() if abs(a) > 1.e-10]

def find_elements_4(A):
    return A[where(0.0 != A)]  # (for where see § VI.3.2)
```

\(^2\)Recall that the shell IPython adds additional functionality to standard Python. These extra functions are called magic functions.