Unit testing with pytest and nose\textsuperscript{1}

Hans Petter Langtangen\textsuperscript{1,2}

\textsuperscript{1}Center for Biomedical Computing, Simula Research Laboratory
\textsuperscript{2}Department of Informatics, University of Oslo

Mar 23, 2015

Contents

1 Requirements of the test function 2
2 Writing the test function; precomputed data 3
3 Writing the test function; exact numerical solution 4
4 Testing of function robustness 4
5 Automatic execution of tests 7

Unit testing is widely a used technique for verifying software implementation. The idea is to identify small units of code and test each unit, ideally in a way such that one test does not depend on the outcome of other tests. Several tools, often referred to as testing frameworks, exist for automatically running all tests in a software package and report if any test failed. The value of such tools during software development cannot be exaggerated. Below we describe how to write tests that can be used by either the nose\textsuperscript{2} or the pytest\textsuperscript{3} testing frameworks. Both these have a very low barrier for beginners, so there is no excuse for not using nose or pytest as soon as you have learned about functions in programming.

\textsuperscript{1}The material in this document is taken from a chapter in the book \textit{A Primer on Scientific Programming with Python}, 4th edition, by the same author, published by Springer, 2014.
\textsuperscript{2}https://nose.readthedocs.org/
\textsuperscript{3}http://pytest.org/latest/
Contents

Model software. We need a piece of software we want to test. Here we choose a function that runs Newton’s method for solving algebraic equations $f(x) = 0$. A very simple implementation goes like

```python
def Newton_basic(f, dfdx, x, eps=1E-7):
    n = 0  # iteration counter
    while abs(f(x)) > eps:
        x = x - f(x)/dfdx(x)
        n += 1
    return x, f(x), n
```

1 Requirements of the test function

The simplest way of using the pytest or nose testing frameworks is to write a set of test functions, scattered around in files, such that pytest or nose can automatically find and run all the test functions. To this end, the test functions need to follow certain conventions.

<table>
<thead>
<tr>
<th>Test function conventions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The name of a test function starts with <code>test_</code>.</td>
</tr>
<tr>
<td>2. A test function cannot take any arguments.</td>
</tr>
<tr>
<td>3. Any test must be formulated as a boolean condition.</td>
</tr>
<tr>
<td>4. An <code>AssertionError</code> exception is raised if the boolean condition is false (i.e., when the test fails).</td>
</tr>
</tbody>
</table>

There are many ways of raising the `AssertionError` exception:

```python
# Formulate a test
tol = 1E-14  # comparison tolerance for real numbers
success = abs(reference - result) < tol
msg = 'computed_result=%d != %d' % (result, reference)

# Explicit raise
if not success:
    raise AssertionError(msg)

# assert statement
assert success, msg
```
# nose tools
import nose.tools as nt
nt.assert_true(success, msg)
# or
nt.assert_almost_equal(result, reference, msg=msg, delta=tol)

This book contains a lot of test functions following the conventions of the pytest
and nose testing frameworks, and we almost exclusively use the plain assert
statement to have full control of what the test method is. In more complicated
software the many functions in nose.tools may save quite some coding and are
convenient to use.

2 Writing the test function; precomputed data

Newton’s method for solving an algebraic equation \( f(x) = 0 \) results in only an
approximate root \( x_r \), making \( f(x_r) \neq 0 \), but \( |f(x_r)| \leq \epsilon \), where \( \epsilon \) is supposed
to be a prescribed number close to zero. The problem is that we do not know
beforehand what \( x_r \) and \( f(x_r) \) will be. However, if we strongly believe the
function we want to test is correctly implemented, we can record the output from
the function in a test case and use this output as a reference for later testing.

Assume we try to solve \( \sin(x) = 0 \) with \( x = -\pi/3 \) as start value. Running
Newton_basic with a moderate-size \( \epsilon \) of \( 10^{-2} \) gives \( x = 0.000769691024206 \),
\( f(x) = 0.000769690948209 \), and \( n = 3 \). A test function can now compare new
computations with these reference results. Since new computations on another
computer may lead to round-off errors, we must compare real numbers with a
small tolerance:

def test_Newton_basic_precomputed():
    from math import sin, cos, pi
    
def f(x):
        return sin(x)
    
def dfdx(x):
        return cos(x)
    
x_ref = 0.000769691024206
f_x_ref = 0.000769690948209
n_ref = 3
    
x, f_x, n = Newton_basic(f, dfdx, x=-pi/3, eps=1E-2)
    
tol = 1E-15  # tolerance for comparing real numbers
assert abs(x_ref - x) < tol  # is x correct?
assert abs(f_x_ref - f_x) < tol  # is f_x correct?
assert n == 3  # is n correct?

The assert statements involving comparison of real numbers can alternatively
be carried out by nose.tools functionality:
For simplicity we dropped the optional messages explaining what went wrong if tests fail.

3 Writing the test function; exact numerical solution

Approximate numerical methods are sometimes exact in certain special cases. An exact answer known beforehand is a good starting point for a test since the implementation should reproduce the known answer to machine precision. For Newton’s method we know that it finds the exact root of \( f(x) = 0 \) in one iteration if \( f(x) \) is a linear function of \( x \). This fact leads us to a test with \( f(x) = ax + b \), where we can choose \( a \) and \( b \) freely, but it is always wise to choose numbers different from 0 and 1 since these have special arithmetic properties that can hide programming errors.

The test function contains the problem setup, a call to the function to be verified, and assert tests on the output, this time also with an error message in case tests fail:

```python
def test_Newton_basic_linear():
    """Test that a linear function is handled in one iteration."""
    f = lambda x: a*x + b
    dfdx = lambda x: a
    a = 0.25; b = -4
    x_exact = 16
    eps = 1E-5
    x, f_x, n = Newton_basic(f, dfdx, -100, eps)
    tol = 1E-15  # tolerance for comparing real numbers
    assert abs(x - 16) < tol, 'wrong root x=%g != 16' % x
    assert abs(f_x) < eps, '|f(root)|=%g > %g' % (f_x, eps)
    assert n == 1, 'n=%d, but linear f should have n=1' % n
```

4 Testing of function robustness

Our Newton_basic function is very basic and suffers from several problems:

- for divergent iterations it will iterate forever,
- it can divide by zero in \( f(x)/dfdx(x) \),
- it can perform integer division in \( f(x)/dfdx(x) \),
- it does not test whether the arguments have acceptable types and values.

A more robust implementation dealing with these potential problems look as follows:
def Newton(f, dfdx, x, eps=1E-7, maxit=100):
    if not callable(f):
        raise TypeError('f is %s, should be function or class with __call__'
                        % type(f))
    if not callable(dfdx):
        raise TypeError('dfdx is %s, should be function or class with __call__'
                         % type(dfdx))
    if not isinstance(maxit, int):
        raise TypeError('maxit is %s, must be int' % type(maxit))
    if maxit <= 0:
        raise ValueError('maxit=%d <= 0, must be > 0' % maxit)
    n = 0  # iteration counter
    while abs(f(x)) > eps and n < maxit:
        try:
            x = x - f(x)/float(dfdx(x))
        except ZeroDivisionError:
            raise ZeroDivisionError('dfdx(%g)=%g - cannot divide by zero' % (x, dfdx(x)))
        n += 1
    return x, f(x), n

The numerical functionality can be tested as described in the previous example, but we should include additional tests for testing the additional functionality. One can have different tests in different test functions, or collect several tests in one test function. The preferred strategy depends on the problem. Here it may be natural to have different test functions only when the \( f(x) \) formula differs to avoid repeating code.

To test for divergence, we can choose \( f(x) = \tanh(x) \), which is known to lead to divergent iterations if not \( x \) is sufficiently close to the root \( x = 0 \). A start value \( x = 20 \) reveals that the iterations are divergent, so we set \( \text{maxit}=12 \) and test that the actual number of iterations reaches this limit. We can also add a test on \( x \), e.g., that \( x \) is a big as we know it will be: \( x > 10^{50} \) after 12 iterations. The test function becomes

```python
def test_Newton_divergence():
    from math import tanh
    f = tanh
dfdx = lambda x: 10./(1 + x**2)
    x, f_x, n = Newton(f, dfdx, 20, eps=1E-4, maxit=12)
    assert n == 12
    assert x > 1E+50
```

To test for division by zero, we can find an \( f(x) \) and an \( x \) such that \( f'(x) = 0 \). One simple example is \( x = 0, f(x) = \cos(x) \), and \( f'(x) = -\sin(x) \). If \( x = 0 \) is the start value, we know that a division by zero will take place in the first iteration, and this will lead to a \( \text{ZeroDivisionError} \) exception. We can explicitly handle this exception and introduce a boolean variable \( \text{success} \) that is \( \text{True} \) if the exception is raised and otherwise \( \text{False} \). The corresponding test function reads
def test_Newton_div_by_zero1():
    from math import sin, cos
    f = cos
    dfdx = lambda x: -sin(x)
    success = False
    try:
        x, f_x, n = Newton(f, dfdx, 0, eps=1E-4, maxit=1)
    except ZeroDivisionError:
        success = True
    assert success

There is a special nose.tools.assert_raises helper function that can be used to test if a function raises a certain exception. The arguments to assert_raises are the exception type, the name of the function to be called, and all positional and keyword arguments in the function call:

import nose.tools as nt

def test_Newton_div_by_zero2():
    from math import sin, cos
    f = cos
    dfdx = lambda x: -sin(x)
    nt.assert_raises(ZeroDivisionError, Newton, f, dfdx, 0, eps=1E-4, maxit=1)

Let us proceed with testing that wrong input is caught by function Newton. Since the same type of exception is raised for different types of errors we shall now also examine (parts of) the exception messages. The first test involves an argument f that is not a function:

def test_Newton_f_is_not_callable():
    success = False
    try:
        Newton(4.2, 'string', 1.2, eps=1E-7, maxit=100)
    except TypeError as e:
        if "f is <type 'float'>" in e.message:
            success = True

As seen, success = True demands that the right exception is raised and that its message starts with f is <type 'float'>. What text to expect in the message is evident from the source in function Newton.

The nose.tools module also has a function for testing the exception type and the message content. This is illustrated when dfdx is not callable:

def test_Newton_dfdx_is_not_callable():
    nt.assert_raises_regexp(
        TypeError, "dfdx is <type 'str'>",
        Newton, lambda x: x**2, 'string', 1.2, eps=1E-7, maxit=100)

Checking that Newton catches maxit of wrong type or with a negative value can be carried out by these test functions:
def test_Newton_maxit_is_not_int():
    nt.assert_raises_regexp(
        TypeError, "maxit is <type 'float'>",
        Newton, lambda x: x**2, lambda x: 2*x,
        1.2, eps=1E-7, maxit=1.2)

def test_Newton_maxit_is_neg():
    nt.assert_raises_regexp(
        ValueError, "maxit=-2 <= 0",
        Newton, lambda x: x**2, lambda x: 2*x,
        1.2, eps=1E-7, maxit=-2)

The corresponding support for testing exceptions in pytest is

    import pytest
    with pytest.raises(TypeError) as e:
        Newton(lambda x: x**2, lambda x: 2*x, 1.2, eps=1E-7, maxit=-2)

5 Automatic execution of tests

Our code for the Newton_basic and Newton functions is placed in a file eq_solver.py to together with the tests. To run all test functions with names of the form test_() in this file, use the nosetests or py.test commands, e.g.:

```
Terminal> nosetests -s eq_solver.py
..........
Ran 10 tests in 0.004s
OK
```

The -s option causes all output from the called functions in the program eq_solver.py to appear on the screen (by default, nosetests and py.test suppress all output). The final OK points to the fact that no test failed. Adding the option -v prints out the outcome of each individual test function. In case of failure, the AssertionError exception and the associated message, if existing, are displayed. Pytest also displays the code that failed.

One can also collect test functions in separate files with names starting with test. A simple command nosetests -s -v will look for all such files in this folder as well as in all subfolders if the folder names start with test or end with _test or _tests. By following this naming convention, nosetests can automatically run a potentially large number of tests and give us quick feedback. The py.test -s -v command will look for and run all test files in the entire tree of any subfolder.

---

4http://tinyurl.com/pwyasaa/tech/eq_solver.py

7
Remark on classical class-based unit testing.

The pytest and nose testing frameworks allow ordinary functions, as explained above, to perform the testing. The most widespread way of implementing unit tests, however, is to use class-based frameworks. This is also possible with nose and with a module unittest that comes with standard Python. The class-based approach is very accessible for people with experience from JUnit in Java and similar tools in other languages. Without such a background, plain functions that follow the pytest/nose conventions are faster and cleaner to write than the class-based counterparts.
Index

nose tests, 1
nosetests command, 7

py.test command, 7
pytest tests, 1

software testing
  nose, 1
  pytest, 1

unit testing, 1