Computational Programming with Python
Unit 14: Tests and profiling

Numerical Analysis, Lund University
Lecturers: Claus Führer, Alexandros Sopasakis

2018
Tests
Why Tests?

- Because you do them *anyway*
- Because it will keep your code *alive*.
Automated Tests

Automated tests

- ensure a constant (high) quality standard of your code
- serve as a documentation of the use of your code
- document the test cases → test protocol
Let’s assume we want to test an implementation of the bisection algorithm:

```python
def bisect(f, a, b, tol=1.e-8):
    if f(a) * f(b) > 0:
        raise ValueError("Incorrect initial [a, b]"")
    for i in range(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')
```
We check a problem with a known solution. Does the code find a zero of \( f(x) = x \)?

```python
def test_identity():
    result = bisect(lambda x: x, -1., 1.)
    expected = 0.
    assert allclose(result, expected), 'expected zero not found'

test_identity()
```

Note the command `allclose`
Example (Cont.)

Does the code handle wrong input correctly?

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

test_badinput()
You will want to *put your tests together* and automatize them:

```python
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()
```
Test results

Here the results with two different tolerance parameters:

**Ran 1 test in 0.002s**

**OK**

... and here with a loose tolerance

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)**
Unittest: Grouping tests together

We recommend to group tests together

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()
unittest.TestCase.assertRaises

Note the method `assertRaises`:

- Exception: the expected exception type
- `bisect`: the function to be called
- `self.identity_fcn, -1.2, 1., tol`: the parameters of this function
Unittest: Preparing tests

Sometimes the execution of tests need some preparation. See special method setUp:

```python
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()
```
... and here the functions which we want to test

class NotFoundError(Exception):
    pass

def find_string(file, string):
    for i, lines in enumerate(file.readlines()):
        if string in lines:
            return i
    raise NotFoundError('String {} not found in File {}'.
        format(string, file.name)
Profiling
The code is slow. Why?

Three possible reasons:

- The problem is big
  *Not much to do about this*
- The algorithm is inherently slow
  *Nor this*
- The implementation is bad (recursive, etc.)
  *We can fix this!*
Finding the slow parts

Different parts of the code are differently fast.

*Profiling* means that we run the program and measure how much time is spent in each function.

A good way to do this in Python is to use the `cProfile` module.
The timeit module

The `timeit` module is ideal for measuring execution time for short statements:
What is faster?

```python
def var1():
    [i**2 for i in range(10000) if i%2==0]

def var2():
    li=[]
    for i in range(10000):
        if i%2==0:
            li.append(i**2)
```

When this code is saved in `timeit_ex.py`, then

```python
t1=ti.Timer('te.var1()','import timeit_ex as te')
t1.timeit(100) # a hundred executions
t2=ti.Timer('te.var2()','import timeit_ex as te')
t2.timeit(100) # a hundred executions
```

gives us the answer....
The timeit module provides a Timer class with the signature

```python
class Timer(stmt [, setup]):
```

where

- `stmt` is a string containing the statement(s) to be executed.
- `setup` is a string with the statement(s) executed beforehand, i.e. not affecting the time measurement.

It has a method `timeit` which takes the numbers of repetitions as argument.
Other timing Methods

For other timing methods consult the course book.