Best practices in scientific programming

Software Carpentry, Part I

Rike-Benjamin Schuppner[1]

Humboldt-Universität zu Berlin
Bernstein Center for Computational Neuroscience Berlin

Python Winterschool Warsaw, Feb 2010

[1] rike.schuppner@bccn-berlin.de

Outline

Collaborating with VCS

Subversion (SVN)

Unittests

Debugging

- `pdb`

Optimisation strategies / profiling

- `timeit`
- `cProfile`

Python tools for agile programming

- I'll present:
  - Python standard 'batteries included' tools
  - no graphical interface necessary
  - magic commands for `ipython`
- Many tools, based on command line or graphical interface
- Alternatives and cheat sheets are on the Wiki

Version Control Systems

- Central repository of files and directories on a server
- The repository keeps track of changes in the files
- Manipulate versions (compare, revert, merge, …)
- How does this look in 'real life'?

Subversion (SVN)

- Create a new repository

  ⇒ `svnadmin create PATH`

  ! *requires security decisions about access to repository, have a look at the SVN book*

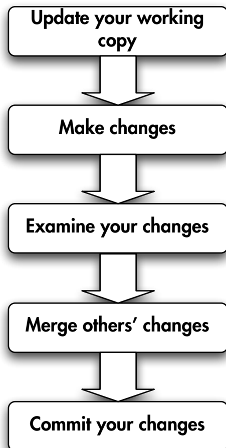
- Get a local copy of a repository

  ⇒ `svn co URL [PATH]`

- Checkout a copy of the course SVN repository

  ⇒ `svn co --username=your_username https://escher.fuw.edu.pl/svn/python-winterschool/public winterschool`

Basic SVN cycle



`svn update`

`svn add` `svn copy`

`svn delete` `svn move`

`svn status` `svn diff` `svn revert`

`svn update`

resolve conflicts, then `svn resolved`

`svn commit -m "meaningful message"`

Time for a demo

SVN notes

- SVN cannot merge binary files $\Rightarrow$ don't commit large binary files that change often (e. g., results files)
- At each milestone, commit the whole project with a clear message marking the event

  $\Rightarrow$ `svn commit -m "submission to Nature"`

- There's more to it:
  - Branches, tags, repository administration
  - Graphical interfaces: subclipse for Eclipse, TortoiseSVN, …
  - Distributed VCS: Mercurial, git, Bazaar

Test Suites in python: `unittest`

- Automated tests are a fundamental part of modern programming practices
- `unittest`: standard Python testing library.

What to test?

- Test general routines with specific ones
- Test special or boundary cass
- Test that meaningful error messages are raised upon corrupt input
  - Relevant when wrtiting scientific libraries

Anatomy of a TestCase

```
1  import unittest
2  class FirstTestCase(unittest.TestCase):
3      def testtruisms(self):
4          """All methods beginning with “ test ” are
              executed"""
5          self.assertTrue(True)
6          self.assertFalse(False)
7
8      def testequality(self):
9          """Docstrings are printed during executions of
              the tests in the Eclipse IDE"""
10         self.assertEqual(1, 1)
11
12 if __name__ == '__main__':
13     unittest.main()
```

TestCase.assertSomething

```
1 assertTrue('Hi'.islower()) => fail
2 assertFalse('Hi'.islower()) => pass
3 assertEqual([2, 3], [2, 3]) => pass
4 assertAlmostEqual(1.125, 1.12, 2) => pass
5 assertAlmostEqual(1.125, 1.12, 3) => fail
6 assertRaises(exceptions.IOError, file, 'inexistent', 'r
     ')       => pass
7 assertTrue('Hi'.islower(), 'One of the letters is not
     lowercase')
```

Multiple TestCases

```
1  import unittest
2
3  class FirstTestCase(unittest.TestCase):
4      def testtruisms(self):
5          self.assertTrue(True)
6          self.assertFalse(False)
7
8  class SecondTestCase(unittest.TestCase):
9      def testapproximation(self):
10         self.assertAlmostEqual(1.1, 1.15, 1)
11
12 if __name__ == '__main__':
13     # execute all TestCases in the module
14     unittest.main()
```

setUp and tearDown

```
 1 import unittest
 2
 3 class FirstTestCase(unittest.TestCase):
 4     def setUp(self):
 5         """setUp is called before every test"""
 6         pass
 7
 8     def tearDown(self):
 9         """tearDown is called at the end of every test
               """
10         pass
11
12     # … all tests here …
13
14 if __name__ == '__main__':
15     unittest.main()
```

Time for a demo

Debugging

- The best way to debug is to avoid it
- Your test cases should already exclude a big portion of possible causes
- Don't start littering your code with 'print' statements
- Core ideas in debugging: you can stop the execution of your application at the bug, look at the state of the variables, and execute the code step by step

pdb, *the Python debugger*

- Command-line based debugger
- `pdb` opens an interactive shell, in which one can interact with the code
  - examine and change value of variables
  - execute code line by line
  - set up breakpoints
  - examine calls stack

Entering the debugger

- Enter at the start of a program, from command line:
  - `python -m pdb mycode.py`
- Enter in a statement or function:

```
1 import pdb
2 # your code here
3 if __name__ == '__main__':
4     pdb.runcall(function[, argument, …])
5     pdb.run(expression)
```

- Enter at a specific point in the code:

```
1 import pdb
2 # some code here
3 # the debugger starts here
4 pdb.set_trace()
5 # rest of the code
```

Entering the debugger

- From ipython, when an exception is raised:
  - %pdb – preventive
  - %debug – post-mortem

Time for a demo

Some general notes to optimisation

- Readable code is usually better than faster code
- Only optimise, if it's absolutely necessary
- Only optimise your bottlenecks

Python code optimisation

- Python is slower than C, but not prohibitively so
- In scientific applications, this difference is even less noticeable (when using numpy, scipy, ...)
  - for basic tasks as fast as Matlab, sometimes faster
  - as Matlab, it can easily be extended with C or Fortran code
- Profiler = Tool that measures where the code spends time

timeit

- precise timing of a function / expression
- test different versions of small amount of code, often used in interactive Python shell

```
1 from timeit import Timer
2
3 # execute 1 million times, return elapsed time(
      sec)
4 Timer("module.function(arg1, arg2)", "import
      module").timeit()
5
6 # more detailed control of timing
7 t = Timer("module.function(arg1, arg2)", "import
      module")
8 # make three measurements of timing, repeat 2
      million times
9 t.repeat(3, 2000000)
```

Time for a demo

cProfile

- standard Python module to profile an entire application (profile is an old, slow profiling module)
- Running the profiler from command line:
  - python -m cProfile myscript.py
  - options -o output_file
  - -s sort_mode (calls, cumulative, name, ...)
- from interactive shell / code:

```
import cProfile
cProfile.run(expression [, "filename.profile"])
```

cProfile, *analysing profiling results*

- From interactive shell / code:

```
1 import pstats
2 p = pstats.Stats("filename.profile")
3 p.sort_stats(sort_order)
4 p.print_stats()
```

- Simple graphical description with RunSnakeRun

`cProfile`, *analysing profiling results*

- Look for a small number of functions that consume most of the time; those are the 'only' parts that you should optimise
- High number of calls per functions
  $\Rightarrow$ bad algorithm?
- High time per call
  $\Rightarrow$ consider caching
- High times, but valid
  $\Rightarrow$ consider using libraries like `numpy` or rewriting in C