From theory to practice: Standard tools
Software carpentry, Part II

Pietro Berkes, Brandeis University
Outline

- Collaborating: SVN
- Profiling: timeit, cProfile
- Debugging: pdb
- Documentation, code clarity: pydoc, pylint
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, …)

Local copy

Central repository

Revision 109

Revision 107

Revision 108

Revision 109

Revision 107

Revision 108
VCS for the lone scientist

- Store source code, data, papers, and presentations about a project
  - Backup
  - Reversible changes
  - Multiple synchronized copies of your project: now you can work from home, too!
VCS for a team of scientists

- Multiple people working at the same time on the same project (software libraries, papers)
  - Handle simultaneous changes to the same files and merge them or handle conflicts
  - Look at recent changes, who is responsible for newest versions, and much more
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 https://portal.bccn-berlin.de/svn/python-summerschool/public`
Basic SVN cycle

Update your working copy
- `svn update`

Make changes
- `svn add`
- `svn delete`
- `svn copy`
- `svn move`

Examine your changes
- `svn status`
- `svn diff`
- `svn revert`

Merge others’ changes
- `resolve conflicts`
- `svn resolved`

Commit your changes
- `svn commit -m “meaningful message”`
SVN notes

- SVN cannot merge binary files => 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
  `svn commit -m"submission to Nature"`

- There’s more to it:
  - Branches, tags, repository administration
  - Graphical interfaces: subclipse for Eclipse, …
  - 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
import unittest

class FirstTestCase(unittest.TestCase):

    def testtruisms(self):
        """All methods beginning with 'test' are executed""
        self.assertTrue(True)
        self.assertFalse(False)

    def testequality(self):
        """Docstrings are printed during executions of the tests in the Eclipse IDE""
        self.assertEqual(1, 1)

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

```assertTrue('Hi'.islower())``` => fail
```assertFalse('Hi'.islower())``` => pass
```assertEqual([2, 3], [2, 3])``` => pass
```assertAlmostEqual(1.125, 1.12, 2)``` => pass
```assertAlmostEqual(1.125, 1.12, 3)``` => fail

```assertRaises(exceptions.IOError, file, 'inexistent', 'r')``` => pass

```assertTrue('Hi'.islower(), 'One of the letters is not lowercase')```
import unittest

class FirstTestCase(unittest.TestCase):
    def testtruisms(self):
        self.assertTrue(True)
        self.assertFalse(False)

class SecondTestCase(unittest.TestCase):
    def testapproximation(self):
        self.assertAlmostEqual(1.1, 1.15, 1)

if __name__ == '__main__':
    # execute all TestCases in the module
    unittest.main()
setUp and tearDown

```python
import unittest

class FirstTestCase(unittest.TestCase):
    
def setUp(self):
        """setUp is called before every test""
        pass

    def tearDown(self):
        """tearDown is called at the end of every test""
        pass

    # ... all tests here ...

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

Pietro Berkes, 9/1/2009
Software carpentry - Part II
Python code optimization

- Python is slower than C, but not prohibitively so
- In scientific applications, this difference is even less noticeable (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 a small amount of code, often used in interactive Python shell

```python
from timeit import Timer

# execute 1 million times, return elapsed time (sec)
Timer("module.function(arg1, arg2)", "import module").timeit()

# more detailed control of timing
T = Timer("module.function(arg1, arg2)", "import module")
# make three measurements of timing, repeat 2 million times
T.repeat(3, 2000000)
```

- in ipython, you can use the `%%timeit` magic command
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, analyzing profiling results

- From interactive shell/code:
  ```python
  import pstat
  p = pstat.Stats("filename.profile")
  p.sort_stats(sort_order)
  p.print_stats()
  ```

- Simple graphical description with RunSnakeRun
cProfile, analyzing profiling results

- Look for a small number of functions that consume most of the time, those are the only parts that you should optimize.

- High number of calls per functions
  => bad algorithm?

- High time per call
  => consider caching

- High times, but valid
  => consider using libraries like numpy or rewriting in C
Debugging

- The best way to debug is to avoid it
- Your test cases should already exclude a big portion of the possible causes
- Don’t start littering your code with `print` statements
- Core idea 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:
  
  ```
  import pdb
  # your code here
  if __name__ == '__main__':
      pdb.runcall(function[, argument, ...])
      pdb.run(expression)
  ```

- Enter at a specific point in the code:
  
  ```
  import pdb
  # some code here
  # the debugger starts here
  pdb.set_trace()
  # rest of the code
  ```
Entering the debugger

- From ipython:
  - `%pdb` - *preventive*
  - `%debug` - *post-mortem*
Two more useful tools

- **pydoc**: creating documentation from your docstrings
  
  `pydoc [-w] module_name`

- **pylint**: check that your code respects standards
The End

- Exercises after the tea break...

```
+---+---+---+
| 1 |   |   |
+---+---+---+
| 2 | 3 |   |
+---+---+---+
| 5 |   | 4 |
+---+---+---+
| 5 | 1 | 4 |
+---+---+---+
| 7 |   | 2 |
+---+---+---+
| 7 | 8 | 9 |
+---+---+---+
| 8 | 7 | 9 |
+---+---+---+
| 4 | 6 | 3 |
+---+---+---+
|   |   | 5 |
+---+---+---+
```
## Test Case methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assert_(expr[, msg])</code></td>
<td><code>assertTrue(isinstance([1,2], list) =&gt; pass</code>)</td>
</tr>
<tr>
<td><code>assertTrue(expr[, msg])</code></td>
<td><code>assertTrue('Hi'.islower()) =&gt; fail</code></td>
</tr>
<tr>
<td><code>assertFalse(expr[, msg])</code></td>
<td></td>
</tr>
<tr>
<td><code>assertEqual(first, second[, msg])</code></td>
<td><code>assertEqual([2, 3], [2, 3]) =&gt; pass</code></td>
</tr>
<tr>
<td><code>assertNotEqual(first, second[, msg])</code></td>
<td><code>assertEqual(1.2, 1.3) =&gt; fail</code></td>
</tr>
<tr>
<td><code>assertAlmostEqual(first, second[, places[, msg]])</code></td>
<td><code>assertAlmostEqual(1.125, 1.12, 2) =&gt; pass</code></td>
</tr>
<tr>
<td><code>assertNotAlmostEqual(first, second[, places[, msg]])</code></td>
<td><code>assertAlmostEqual(1.125, 1.12, 3) =&gt; fail</code></td>
</tr>
<tr>
<td><code>assertRaises(exception, callable, ...)</code></td>
<td><code>assertRaises(exceptions.IOError, file, 'inexistent', 'r') =&gt; pass</code></td>
</tr>
<tr>
<td></td>
<td><code>assertRaises(exceptions.SyntaxError, file, 'inexistent', 'r') =&gt; fail</code></td>
</tr>
<tr>
<td><code>fail([msg])</code></td>
<td><code>fail() =&gt; fail</code></td>
</tr>
</tbody>
</table>