Agile development

Agile development work cycle

Repeat until all features are implemented:

1. Write a test that defines your next feature
2. Write the simplest version of code that makes your test pass
3. Run the tests and debug until all tests pass
4. Refactor (remove duplication, reorganize the code)
5. Go back to 3 until necessary

If speed or memory are an issue:

6. Optimize only at this point
7. Go back to 3 until necessary

Reacting to bugs

1. Use debugger to isolate bug
2. Write test case that reproduces bug
3. Correct the bug
4. Check that *all* tests pass

Software carpentry tools

pyflakes – Static analysis of Python code

`pyflakes path`	Analyze all the Python files in `path` and subdirectories, checking for errors. The module is not executed, don't worry about side effects.

pep8 – Static analysis of Python code

`pep8 path`	Analyze all the Python files in `path` and subdirectories, reporting the parts of the code that do not comply with the Python style standard, PEP8.

flake8 – Static analysis of Python code

`flake8 path`	Combined power of `pyflakes` and `pep8`.

pydoc – Generate documentation pages from Python docstrings

`pydoc module_name`	text output
`pydoc -w module_name`	html output
`pydoc -g`	open graphical interface

coverage.py – Code coverage for testing

Execute `my_program.py` with arguments `arg1` and `arg2` with plain coverage analysis:

```
coverage run my_program.py arg1 arg2
```

With branch coverage analysis:

```
coverage run --branch my_program.py arg1 arg2
```

Coverage information is saved in a file called `.coverage`, or as specified in the environment variable `COVERAGE_FILE`.

Print coverage report (`-m` show the lines that were not executed):

```
coverage report -m
```

Generate HTML coverage report:

```
coverage html -d html_directory
```

Remove coverage information:

```
coverage erase
```

Generate annotated version of source code:

```
coverage annotate
```

Legend:

```
> executed
! missing (not executed)
-  excluded
```

unittest

Basic structure of a test suite

```
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,
        even if an exception was raised. """
        pass

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

    # ... more tests here ...


class SecondTestCase(unittest.TestCase):

    def test_approximation(self):
        self.assertAlmostEqual(1.1, 1.15, 1)


if __name__ == '__main__':
    # run all TestCase's in this module
    unittest.main()
```

define if necessary to create fixtures

Assert methods in unittest.TestCase

Most *assert* methods accept an optional *msg* argument, which is printed in case the assertion fails to facilitate debugging. A complete list of all available *assert* methods is available at http://tinyurl.com/cmohfrc **.** Most methods also have a negated counterpart, e.g. `assertEqual` and `assertNotEqual` .

`assert_(expr[, msg)` `assertTrue(expr[, msg])`	Fail if *expr* is False
`assertFalse(expr[, msg])`	Fail if *expr* is True
`assertEqual(first, second[, msg])`	Fail if *first* is not equal to *second*
`assertAlmostEqual(first, second [, places[, msg]])`	Fail if *first* is equal to *second* up to the decimal place indicated by *places* (default: 7)
`assertRaises(exception, callable, ...)`	Fail if the function *callable* does not raise an exception of class *exception*. If additional positional or keyword arguments are given, they are passed to *callable*.
`assertRegexpMatches(text, regexp)`	Fail if *text* does not match the regular expression *regexp*
`assertGreater(a, b)`	Fail if *a* smaller or equal to *b*
`assertLess(a, b)`	Fail if *a* greater or equal to *b*
`assertIn(value, sequence)`	Fail if *value* is not an element of *sequence*
`assertIsNone(value)`	Fail if *value* is not None
`assertIsInstance(obj, cls)`	Fail if *obj* is not an instance of *cls*
`assertItemsEquals(actual, expected)`	Fail if the members of *actual* are not equal to the members of *expected* (order is ignored)
`assertDictContainsSubset(subset, full)`	Fail if the entries in dictionary *subset* are not a subset of those in dictionary *full*.
`fail([msg])`	Always fail

cProfile

Invoking the profiler

From the command line:

```
python -m cProfile [-o output_file] [-s sort_order] myscript.py
```

`sort_order` is one of `'calls'`, `'cumulative'`, `'name'`, …
(see cProfile documentation for more)

From interactive shell / code:

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

From ipython:

```
%prun -D<filename> statement
%run -p [profiler_options] myscript.py
```

Looking at saved statistics

From interactive shell / code:

```
import pstat
p = pstat.Stats("filename.profile")
p.sort_stats(sort_order)
p.print_stats()
```

Simple graphical description (needs RunSnakeRun):

```
runsnake filename.profile
```

line_profiler

Profiles selected functions, one line at the time:

1) Decorate all functions that you want to profile with `@profile`;
2) From the command line, run:

```
kernprof -v -l filename.py
```

timing

Execute expression one million times, return elapsed time in seconds:

```
from timeit import Timer
Timer("module.function(arg1, arg2)", "import module").timeit()
```

For a more precise control of timing, use the *repeat* method; it returns a list of repeated measurements, in seconds:

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

In ipython:

`%timeit -n<N> -r<R>` *statement*

Time the execution of *statement*, executing it <N> times (default: adapt to speed of statement). The operation is repeated <R> times, and the best run is reported.

`%time` *statement*

Execute statement once and report CPU and wall clock time

pdb

Invoking the debugger

Enter at the start of a program, from the command line:

```
python -m pdb mycode.py
```

Enter in a statement or function:

```
import pdb
# your code here
if __name__ == '__main__':
    # start debugger at the beginning of a function
    pdb.runcall(function[, argument, ...])
    # execute an expression (string) under the debugger
    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
```

In ipython:

`%pdb` enter the debugger automatically after an exception is raised

`%debug` enter the debugger post-mortem where the exception was thrown

`%run -d -b<L> myscript.py`
execute the script and enter the debugger and at line <L>

Debugger commands

h (help) [***command***]	print help about *command*
n (next)	execute current line of code, go to next line
c (continue)	continue executing the program until next breakpoint, exception, or end of the program
s (step into)	execute current line of code; if a function is called, follow execution inside the function
l (list)	print code around the current line
w (where)	show a trace of the function call that led to the current line
p (print)	print the value of a variable
q (quit)	leave the debugger
b (break) [***lineno*** | ***function***[, ***condition***]]	set a breakpoint at a given line number or function, stop execution there if *condition* is fulfilled
cl (clear)	clear a breakpoint
! (execute)	execute a python command
<enter>	repeat last command