Agile development cheat sheet

Agile development work cycle

1. Write tests that define your application
2. Write simplest version of the code
3. Run the tests and debug until all tests pass
4. Optimize only at this point
5. Go back to 3 until necessary

Reacting to bugs

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

Implementing new features

1. Write tests for new features
2. Write new features in the simplest possible way
   (follow the agile development work cycle)
3. Refactor

SVN cheatsheet

Check-out an SVN repository

```
svn co URL [PATH]
```

Basic work cycle

1. Update your working copy:
   ```
   svn update
   ```

2. Make changes
   ```
   svn add
   svn delete
   svn copy
   svn move
   ```

3. Examine your changes
   ```
   svn status
   svn diff
   svn revert
   ```

4. Merge others' changes into your working copy
   ```
   svn update
   svn resolved
   ```

5. Commit your changes
   ```
   svn commit –m"meaningful message"
   ```

Miscellaneous tools cheatsheet

pydoc

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

pylint

`pylint`	display very long list with all options
`pylint filename.py`	check file for consistency with standards
`pylint module`	check module

unittest cheatsheet

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"""
        pass

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

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

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

Assert methods in unittest.TestCase

Most *assert* methods accept an optional *msg* argument, which is used as an explanation for the error.

`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*
`assertNotEqual(first, second[, msg])`	Fail if *first* is 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)
`assertNotAlmostEqual(first, second [, places[, msg]])`	Fail if *first* is not 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*.
`fail([msg])`	Always fail

cProfile cheatsheet

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"])
```

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
```

timeit cheatsheet

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)
```

pdb cheatsheet

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

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