Software Carpentry Exercises

Before you start:

– create a suitable directory for this exercise
– initialise a git repository inside the new directory
– download the file `maxima.py` from http://www.physik.uzh.ch/~python/python/lecture3/
– add the file to your repository

Remember to commit every significant change to the git repository with a meaningful message.

1 Writing a test suite [basic]

Goals: Write a test suite using the `unittest` module.

Write a test suite, `test_center.py`, that tests the function `center` in the module `string` (http://docs.python.org/library/string.html#string.center). At each step, run the tests and make sure they pass (insert a call to `unittest.main()` at the end of the script as shown in the slides).

In the suite, write three test cases:

a) The first test case checks the functionality of the function, leaving the argument `fillchar` set to its default value. Control that the function works as advertised for

– odd and even widths
– a width smaller than the length of the string
– an empty input string
– a string containing spaces to either extremity

Test that the length of the returned string is correct and that it looks like you expect it to.

Hint: when the number of spaces to be added is odd, there are two possible ways to centre a string. The docstring does not specify which one is correct, so you should test that the returned string is one **or** the other.

b) The second test case checks the functionality of `string.center`, with `fillchar` set to specific values. Test using a letter, a numerical value, and the default value.

c) Finally, test that `string.center` raises a `TypeError` when `fillchar` is set to an empty string, and to a string longer than one character.

2　Deceivingly simple function [intermediate]

Goals: General practice of debugging and unit testing using agile development techniques.

Enter the directory `deceivingly_simple`. The file `maxima.py` contains a function, `find_maxima`, that finds local maxima in a list and returns their indices. Please read the last sentence again: it returns the **indices**, not the values ;–)

a) Using `ipython`, test the function with these input arguments and others of your own invention until you are satisfied that it does the right thing for typical cases (remember that the function returns the indices of the maxima):

```
x = [0, 1, 2, 1, 2, 1, 0]
x = [-i**2 for i in range(-3, 4)]
x = [numpy.sin(2*alpha) for alpha in numpy.linspace(0, 5., 100)]
```

b) Now try with the following inputs:

```
x = [4, 2, 1, 3, 1, 2]
x = [4, 2, 1, 3, 1, 5]
x = [4, 2, 1, 3, 1]
```

For each bug you find, solve it using the agile programming strategy:

  i. Isolate the bug using a debugger
  ii. Write a new test case that reproduces the bug. Try to make the test case as simple as possible; here, this means using the simplest input data that still triggers the bug
  iii. Correct the code
  iv. Make sure that all the tests pass

c) Run a coverage analysis on the tests; there should be at least one statement that is not covered. Write a test that covers it and debug the code

d) You may think that the code is now clean and robust... Look at the output of the function for the input list

```
x = [1, 2, 2, 1]
```

Does the output correspond to your intuition? Think about a reasonable default behaviour in this situation, and meditate on how such a simple function can hide so many complications

e) Implement the "reasonable behaviour" you conceived in item (d) and document it in the docstring, adding a new doctest. Make sure that your function handles these inputs correctly (include them in the tests):

```
x = [1, 2, 2, 3, 1]
x = [1, 3, 2, 2, 1]
x = [3, 2, 2, 3]
```

3   k-Means and numerical fuzzing [advanced]

Goals: Implement a learning algorithm and test it using numerical fuzzing techniques.

k-means clustering is a simple method to assign n-dimensional data points to k groups. Each group is represented by the its mean value and data points are assigned to the closest mean. Figure 1 show a two-dimensional example. A longer explanation can found on http://en.wikipedia.org/wiki/K-means_clustering.
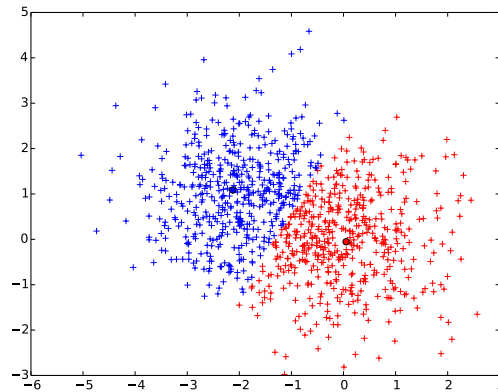

Figure 1: Example of k-Mean clustering in two-dimensions with $k = 2$. Means are at $(0, 0)$ and $(-2, 1)$

A straight forward approach to k-means clustering is to split the algorithm into two functions.

  – A *classification* function that assigns data points to given mean values.
  – A *means* function that calculates new mean values based on a given classification.

The optimal position of the means can then be determined by alternatingly calling these two functions. Once classifying the data points according to the present means and once determining new mean values based on the assigned data points.

  a) Use the Test Driven Development methodology to implement this algorithm. You can start with simple manually entered test data, but should also include tests with randomly generated data.

  b) Check that your code adheres to Python standards using pep8:

         pep8 kmeans.py

    Improve your code until pep8 is happy .

  c) **Scipy** provides k-mean clustering in `scipy.cluster.vq`: http://docs.scipy.org/doc/scipy/reference/cluster.vq.html. Once you are satisfied with your code, compare the results to the **Scipy** implementation.

  d) Profile your code. Save the profile results and examine it using runsnake.