# Test, Debug, Profile Exercises

July 9, 2024

based on exercises from Pietro Berkes


Before you start: To practice good software development workflows, use git while working on these exercises. To setup:

- create a suitable directory for this exercise
- initialise a git repository inside the new directory
- download the files `cleanup.py` and `maxima.py` from the lecture webpage.
- add the files 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.*

The file `cleanup.py` contains a function, `clean` to remove or replace unwanted characters in an input string. Now create a file `test_clean.py` with a test suite for this function. Make sure your tests check for the behaviour as documented in the docstring.

In the suite, write five test cases with one hardcoded pair of input and expected output each:

a) Test a general input with upper and lowercase letters, numbers and symbols (e.g. `H3l1o W0r1d!`).

b) Use a different `allow`-list (e.g. only digits) on the same or a different input.

c) Write a test where you genearte random inputs containing only characters from the allow list. Make sure the inputs are returned unmodified.

d) Pass a single character string to `replace` and check that illegal characters are replaced correctly.

e) Finally, test that `cleanup` raises a `ValueError` as documented, if `replace` contains more then one element.

Once your tests are ready, adjust the code of `clean` and try to introduce a bug for each test. Check that your test suite really catches the problems. You can also try to find bugs your test suite does not detect yet and extend the test suite to cover these cases as well.

## 2 Deceivingly simple function [intermediate]

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

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 https://en.wikipedia.org/wiki/K-means_clustering.

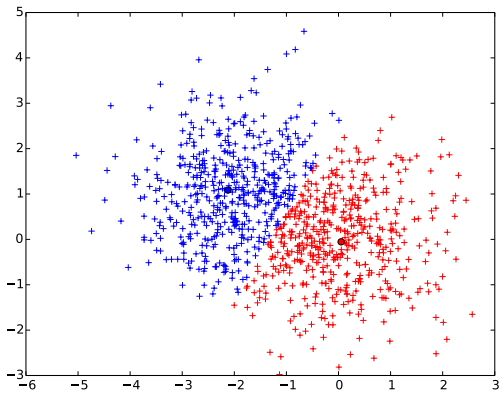


Figure 1: Example of k-Mean clustering in two-dimensions with $k = 2$. Means are at $(0, 0)$ and $(-2, 1)$

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 `pycodestyle`:

```
python3 -m pycodestyle kmeans.py
```

Improve your code until the checker is happy .

c) `Scipy` provides k-mean clustering in `scipy.cluster.vq`: https://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 in your profiler of choice and examine the results.