Advanced Scientific Programming in Python - Summer School
Testing, debugging, profiling, and friends
Pietro Berkes; Sept, 2016

Exercises: Software carpentry

While you are solving the exercises, remember to commit every significant change to your git repository with a meaningful message!

Exercise 1 – Writing a test suite [basic]

Goals:Write a test suite using `pytest` .

Write a test file, `test_center.py`, that tests the method `center` of string objects (https://docs.python.org/3/library/stdtypes.html#str.center). At each step, run the tests and make sure they pass.

In the file, 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
   1. odd and even widths
   2. a width smaller than the length of the string
   3. an empty input string
   4. 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 center 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 the method `center`, with *fillchar* set to specific values. Test using a letter, a numerical value, and the default value.

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

Exercise 2 – Testing with `numpy` and numerical fuzzing [basic]

Goals: Use the `numpy.testing` utility functions and numerical fuzzing techniques to test numerical code.

Write a new test suite, `test_multinomial.py`, to test the function `numpy.random.multinomial` (https://tinyurl.com/448f5dz).

a) Read the documentation and play with the function using `ipython` until you are sure you understand how it works (always leave the *size* argument to its default value).
b) Write a first test case, testing the function in deterministic cases:
    1. when one of the entries has probability 1.0 and the others 0.0, the returned samples must consist only of the entry with probability mass
    2. when one of the entries has probability 0.0, it must not appear in the returned samples
c) Write a numerical fuzzing test case that verifies that, with a large number of samples, the sampling frequency of each entry is close to its probability.

Exercise 3 – 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(0.0, 5.0, 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
    (continues on next page)
    iii. Correct the code
    iv. Make sure that all the tests pass

c) 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 behavior in this situation, and meditate on how such a simple function can hide so many complications

d) *(optional)* Implement the "reasonable behavior" you conceived in c), adding a new test.

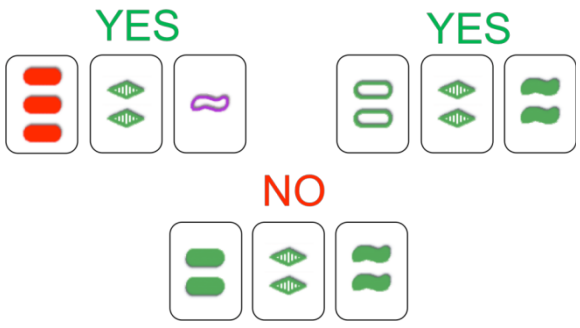
Make sure that your function handles these inputs correctly (include them in the tests):

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

Exercise 4 – The game Set® [advanced]

Goals: Write a solver for the game Set and optimize it until it flies

Set is a logic game consisting in a deck of cards that vary along 4 dimensions: color, shape, texture, and number. For each dimensions, there are 3 possible features (e.g., there are 3 possible textures: full, empty, striped). A valid *set* is formed by three cards that have on each dimension either the *same* feature, or *three different* features. So for example in the image below, the first three cards are a valid set, as they are different in all features across all dimensions; the second three cards also form a valid set, because they share the same features for color and number, and are different in shape and texture; the cards on the bottom are not a set, because two cards have the "full" texture, while one is striped.



In the solitary version of the game, 12 random cards are put on the table, and the player has to find as many valid sets as possible. To test that you understand the rules, visit https://www.nytimes.com/ref/crosswords/setpuzzle.html and solve the daily puzzle (but don't get too distracted!). A longer description of the rules is available at http://www.setgame.com/set/index.html.

In the code, we are going to represent each card by a 4-dimensional vector (for color, shape, texture, and number); each element is either 0, 1, or 2, representing the three possible features for each dimension. For example, two cards might be represented as `[2, 2, 0, 1]` and `[2, 0, 0, 0]`; this means that they have the same features for dimensions 0 and 2 and different features for dimensions 1 and 3.

Enter the directory `set`.

a) The test module `test_set.py` contains a test, `test_is_set,` for a function that takes a list of cards and three indices and returns True if the cards at those indices form a set. Implement `is_set` in `set_solver.py` .

b) The test module also contains a test for a solver that finds all possible sets in a list of cards. Write a brute-force Set solver, `find_sets`: cycle through all possible triplets and call `is_set` for each triplet. If it is a set, append the indices of the cards to a list. Return the list.

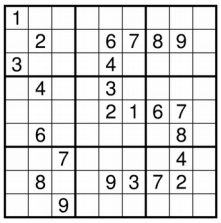
c) The brute-force approach is brutally inefficient. Write a faster version, `find_sets_fast`, using list comprehensions and the function `combinations` from the module `itertools` (https://docs.python.org/3/library/itertools.html). Test the new function using fuzzing: generate random cards and test that the output of `find_sets_fast` is the same of the brute force solver. (Use the function `random_cards` in `set_solver.py` to generate random draws of cards.)

d) Use `timeit` to measure the increase in speed.

e) Given any two cards, there is one and only one card that makes them form a valid set. Use this idea to write a much faster Set solver, and measure its performance.

Exercise 5 - Sudoku solver [advanced]

Goals: Use your new toolbox to develop a Sudoku solver!



Enter the directory `sudoku`. If you don't know what Sudoku is (really?), have a look at http://en.wikipedia.org/wiki/Sudoku .

a) Look at the test cases in `test_sudoku.py` . Write a module `sudoku.py` that makes the tests pass (this is equivalent to writing a Sudoku solution verifier and a Sudoku solver).

Some hints:

- The file `problems.py` contains two dictionaries with Sudoku boards and their solutions. Each board is represented as a 2D list. Write three helper functions, `get_row(grid, nr)`, `get_column(grid, nr)`, and `get_box(grid, nr)`, that return the *nr*-th row, column or box of the Sudoku grid. These will come very handy. Make sure you write tests for the new functions!
- Start by working on the Sudoku verifier, `sudoku.is_solution` .
- Use a brute-force approach to solve the Sudoku board in `sudoku.solve_sudoku`:
  - Start from the first empty cell in the grid
  - Starting from 1, test all digits and check if they violate the constraints; if not, proceed to the next empty cell
  - If none of the digits is allowed in a given cell, leave it blank and go back one cell, incrementing its value by one
  - Continue until the whole grid is filled

  More information about the brute force approach is available on Wikipedia at http://tinyurl.com/mebxw4

b) Profile your code on the `hard2` problem. Examine the results and discuss what could be optimized and how.