Advanced Scientific Programming in Python Summer School, Day 1
Lecturer: Pietro Berkes; Oct 4, 2010

Exercises: Software carpentry

Before you start: download the file `day1_exercises.zip` from https://portal.g-node.org/python-autumnschool/materials:software_carpentry and unzip it in your repository.

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

Exercise 1 – Deceivingly simple function

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

Enter the directory `exercise1`. The file `maxima.py` contains a function, `find_maxima` , that finds local maxima in a list.

a) Run the example code in the docstring of the function using `doctest`, and make sure they pass

b) 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:

```
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, 2*3.14, 100)]
```

c) 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

d) 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 if necessary (it is)

e) So you 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 behavior in this situation, and meditate about how such a simple function can hide so many complications

f) *(optional)* Implement the "reasonable behavior" you conceived in e) 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]
```

Exercise 2 - Bug hunt

Goals: An interesting detective case for bug hunters…

Enter the directory `exercise2`.

a) Have a look at the files `working.data` and `not_working.data`. The program `convert_to_dict.py` reads these files line by line and converts each line into a dictionary of numbers (look at the docstrings in the code for details). Run the program on the two data files
```
    python convert_to_dict.py working.data
    python convert_to_dict.py not_working.data
```
   and observe how it fails to convert the data in the second case

b) Use `pdb` to inspect the code and isolate the bug.

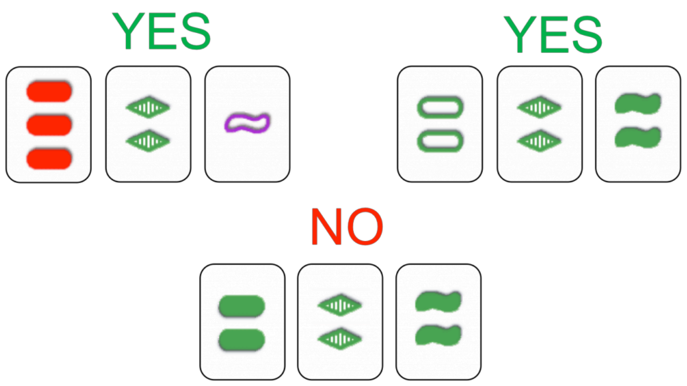
c) In a new file, write a test case that reproduces the error. Run the test and verify that it fails

d) Correct the bug and verify that the new code passes the tests

Exercise 3 – The game Set®

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 http://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` (http://docs.python.org/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 4 - Sudoku solver

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 notes:

   - 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. Save the profile results in `sudoku.profile` . Examine the results and discuss what could be optimized and how (write the response in `optimize.txt`)

c) Check that your code adheres to Python standards using pylint:

```
pylint sudoku.py
```

Improve your code until the overall pylint score is greater than 7.0 .

d) Create documentation for your Sudoku solver:

```
pydoc -w sudoku
```