

Additional Exercise

from Pietro Berkes, updated by Nicola Chiapolini

July 10, 2026

Before you start:

- download the file `carpentry_exercises.zip` from <http://www.physik.uzh.ch/~python/python/lecture3/>
- unzip it into a suitable directory
- create a git repository and add all files

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.

Pretend you have just written the string function `center` from the module `string` (<https://docs.python.org/3/library/stdtypes.html#str.center>). Now write a test suite, `test_center.py`, that tests this function and makes sure it works as documented. 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:

- 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.

- 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.
- 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 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` (documented at <https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.multinomial.html>).

- 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:
 - 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
 - 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.

3 The game Set[®] [advanced]

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

Set is a logic game consisting in a deck of cards that vary along 4 dimensions: colour, 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.

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/crosswords/game/set/> and solve the daily puzzle (but don't get too distracted!).

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.

4 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 <https://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`:
 - i. Start from the first empty cell in the grid
 - ii. Starting from 1, test all digits and check if they violate the constraints; if not, proceed to the next empty cell
 - iii. If none of the digits is allowed in a given cell, leave it blank and go back one cell, incrementing its value by one
 - iv. Continue until the whole grid is filled

More information about the brute force approach is available on Wikipedia at https://en.wikipedia.org/wiki/Sudoku_solving_algorithms

- b) Check that your code adheres to Python standards using `pycodestyle`:

```
python3 -m pycodestyle sudoku.py
```

Improve your code until the checker is happy.

- c) Profile your code on the `hard2` problem. Save the profile results in `sudoku.profile`. Examine the results and discuss what could be optimised and how.