PHOT 110: Introduction to programming

Exercises 15: Testing, Debugging and Performance

Michaël Barbier, Spring semester (2024-2025)

1. Writing unittests using pytest

Unit-tests can be used to verify the correctness of the output of functions with given parameters. There are multiple packages for this, we use the pytest package because of its simplicity. Before you continue, install the `pytest` package (`pip install pytest`).

- Copy the following code of a module that calculates the roots of a linear equation: $ax + b = 0$ or a quadratic equation: $ax^2 + bx + c = 0$. Remember that the roots for a quadratic equation are given by:

$$x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Name the file `module_roots`

```
from math import sqrt

def roots_lin(a, b):
    if a != 0.0:
        x = -b / a
    else:
        x = None
    return x

def roots_quad(a, b, c):
    xp = (-b + sqrt(b**2 - 4*a*c) ) / (2 * a)
    xm = (-b - sqrt(b**2 - 4*a*c) ) / (2 * a)
    return xm, xp
```

- Then create a file with a test class for this module which you name: `test_module_roots.py` and contains the following code:

```
import module_roots as mr

class Test_Module_Roots:
    def test_roots_lin_a_is_zero(self):
        assert mr.roots_lin(0,5) is None

    def test_roots_lin(self):
        assert mr.roots_lin(1,0) == 0.0

    def test_roots_quad(self):
        assert mr.roots_quad(1,0, -1) == (-1.0, 1.0)

    def test_roots_quad_one_solution(self):
        assert mr.roots_quad(2,0, 0) == 0.0
```

- Go to the directory of your code in the terminal and run the command `pytest`.
- The output should indicate there is one test failing: `test_roots_quad_one_solution()`. Why is that?
- Adapt the code in the module such that this test does not fail.

2. Timing a function

Copy/create the following function which creates combination of specific characters and numbers.

```
import random
import timeit

def generate_numbers_1():
    data = []
    for n in ("M","C","S","F"):
        for m in (3, 4, 6, 2, 5, 9, 1):
            data.append( random.gauss(0.0, 1.0) )
    return data

def generate_numbers_2():
    ...

if __name__ == "__main__":
    N = 100000
    time_data = timeit.timeit(generate_numbers_1, number=N)
    print(f"Using a for-loop: {time_data/N} seconds")
```

- Run the code, then adapt the code to get timings for both functions.
- Add the code in `generate_numbers_2` such that it performs the same task but using list comprehension.
- What is faster: using list comprehension or using a for loop?
- See below for example timings:

```
Using a for-loop: 1.1830026979987451e-05 seconds
Using list comprehension: 1.279874159990868e-06 seconds
```

3. Performance testing

The following code can be used to test the performance of a block of code (in this case the function called `main()`). The code writes the time statistics to the terminal but (see the last line) also saves them to the file: "profile_stats.prof". This file can afterwards be visualized by `snakeviz` by typing: `snakeviz profile_stats.prof` in the terminal of pycharm.

```
import cProfile, pstats
import time
import random
import numpy as np

def main():
    generate_numbers_1()
    generate_numbers_2()


if __name__ == "__main__":

    profiler = cProfile.Profile()
    profiler.enable()

    main()

    profiler.disable()
    stats = pstats.Stats(profiler).sort_stats("cumtime")
    stats.print_stats()
    stats.dump_stats("profile_stats.prof")
```

- Add both manners of random number generation of previous exercise to the main function.
- Then run the script and verify whether you can find the "profile_stats.prof" file, and check its content.

- Use snakeviz (install this package if required) by executing the following command on the terminal:

```
snakeviz profile_stats.prof
```