40.317 Lecture 4

Dr. Jon Freeman

28 May 2020

Slido Event #N290

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Agenda

- Speeding up Python code

Speeding up Python Code: Motivation

Why are we studying this topic?

- To understand the steps we should follow to speed up code in *any* language.
- To present performance pitfalls and speedup techniques unique to Python.

Speeding up Python Code

Best overall references:

- http://pypy.org/performance.html
- http://earthpy.org/speed.html

Next, we describe a general eight step approach.

Speeding up Code in 8 Steps

1. Design, part 1: Ask yourself what you are really building (!).
   - E.g. if what you are building is a message-passing system, then pass messages already!

Speeding up Code in 8 Steps

2. Design, part 2: Select your algorithms and data structures before you start coding.
   - Examples:
     - The famous story from Gauss's childhood
     - The 3x3 magic square problem on hackerrank.com
   - An understanding of complexity (c.f. Lecture 2) will really help you here
   - Using less space often translates into taking less time

Speeding up Code in 8 Steps

3. Do not attempt to speed up your code unless you have a clear justification.
   - “Premature optimisation is the root of all evil.” —Donald Knuth

4. Before attempting any speedups, write a regression test which is so complete you can modify your code with confidence.
   - Reference and recommended packages for writing tests: http://docs.python-guide.org/en/latest/writing/tests/

Speeding up Code in 8 Steps

5. Use a (statistical) profiler to determine exactly where the actual performance problem resides.
   - It must be very non-intrusive to avoid distorting the statistics it's gathering.
   - The owners of PyPy recommend vmprof:
     - Supports MacOS X, Windows, and Linux
     - Supports multi-threaded applications
     - Lets you profile only a portion of your code
     - Install via `pip install vmprof` (on Windows, requires Microsoft Visual Studio Build Tools)

Speeding up Code in 8 Steps

6. Look for opportunities to apply concurrent programming (discussed next week): can you divide up the task, run the sub-tasks on separate cores / PCs, and then quickly combine the results?

   There are several popular packages to consider for this purpose, such as:

   – multiprocessing
   – Dask
   – Disco

Speeding up Code in 8 Steps

7. Look for opportunities to apply dynamic programming, often implemented via memoisation.
   - A second example from hackerrank.com
   - An example from Finance: translating one type of security ID to another
     - c.f. “FIGIs”, https://www.openfigi.com

Speeding up Code in 8 Steps

8. Focus your remaining efforts on speeding up the insides of loops, e.g.:
    - Perform all validations / assertion checks in an earlier step
    - Minimise *explicit* branching
        - Make use of polymorphic class methods (discussed in about two weeks)
        - Create a dictionary which maps possible values (known only at runtime) to functions

Python-Specific Tips

First, two meta-comments:

- “Python” is a *specification*. Its default / reference implementation, CPython, happens to be the slowest of the most popular implementations.

- What Python speeds up the most is *your productivity* as a developer.
  - Example: pythonic_radix_sort.py

Python-Specific Tips

- Python code run in CPython is much slower than a corresponding C version. My suggestions, from most general to least general:
  - Run your slowest programs in PyPy, a “just-in-time” compiler, or push speed-critical code into C via Cython
  - Use Numba, designed with the NumPy / SciPy stack in mind
  - Use NumExpr, specifically for speeding up NumPy operations

Python-Specific Tips

- Do not write your own versions of existing functions
- Do not use global variables inside loops
  - Python accesses global variables slowly compared to local variables
  - Either eliminate them or make local copies
- Avoid the use of “dots” inside a loop
  - Python resolves function addresses slowly
  - Use local variables to eliminate function resolution inside loops
- Lessen memory footprint with generators
  - Not the same thing as using less space

Python-Specific Tips

- Eliminate as many explicit loops as possible. Replace them with either:
  - comprehensions, e.g.

    ```
    {w.capitalize(): len(w) for w in words}
    ```

  - or set operations, e.g. replace

    ```
    in_common = set()
    for x in a:
        for y in b:
            if x == y:
                in_common.add(x)
    ```

  - with

    ```
    in_common = set(a) & set(b)
    ```

Thank you

A BETTER WORLD BY DESIGN.

SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN