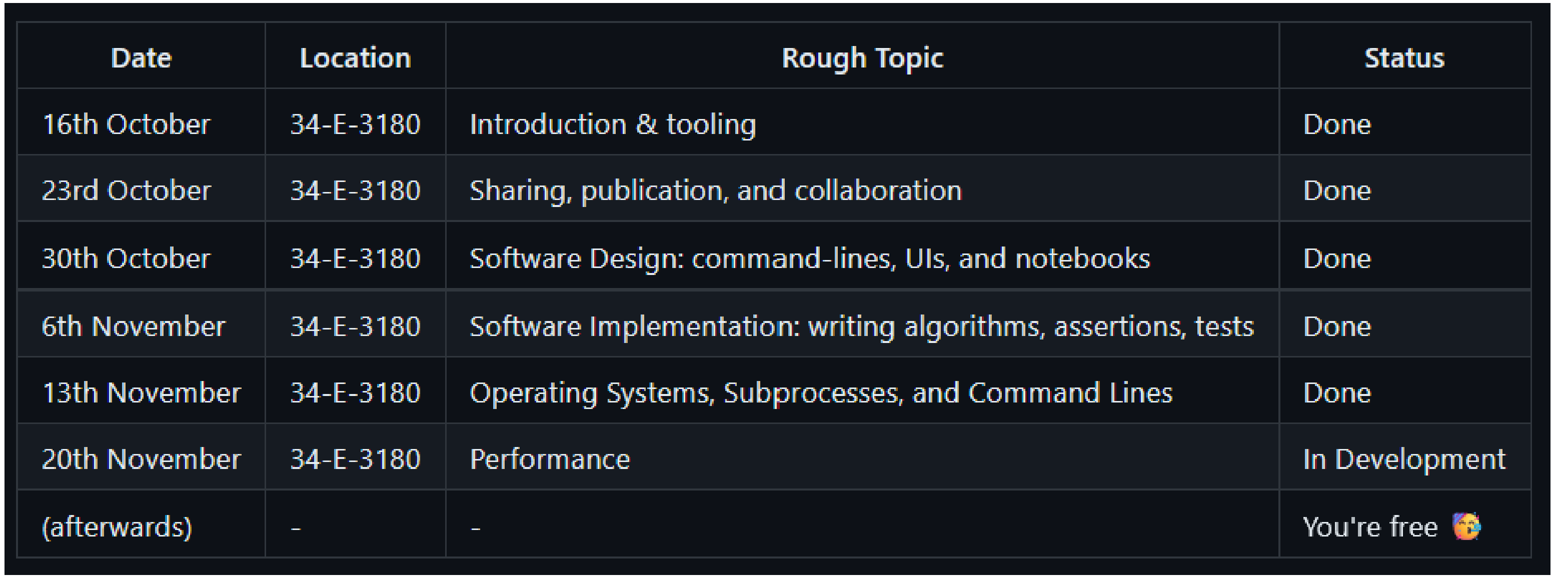
Research Software Engineering Meetups: Performance

2023/11/20

Overview

- Schedule (there isn’t one after this ;))
- Why is performance important?
- Performance Overview
- Code analysis, practical techniques, etc.
- Practical: finding hotspots the fast way
- Practical: profilers
- Overall meetup conclusions

Schedule (provisional)



Date	Location	Rough Topic	Status
16th October	34-E-3180	Introduction & tooling	Done
23rd October	34-E-3180	Sharing, publication, and collaboration	Done
30th October	34-E-3180	Software Design: command-lines, UIs, and notebooks	Done
6th November	34-E-3180	Software Implementation: writing algorithms, assertions, tests	Done
13th November	34-E-3180	Operating Systems, Subprocesses, and Command Lines	Done
20th November	34-E-3180	Performance	In Development
(afterwards)	-	-	You're free 🥳

***source**: https://github.com/adamkewley/rse-meetups/*

Introduction

- You’ve got the tools (**1. tooling**)
- You’ve got a platform (**2. sharing, collaboration**)
- You have an idea of what you want to build (**3. design**)
- Maybe you’re in the process of developing it (**4. implementation**)
- Maybe you’ve had to integrate various other applications (**5. subprocesses**)
- And now you’re finding that it takes forever to run… (**6. performance**)

Performance

(questions for the group)

(1) Why is performance important?

(2) What techniques can be used to improve performance?

(1) – Because (RACE CAR NOISES)	(1) Because it will make my code cleaner	(2) Use a faster programming language
(1) Because it will make my workflow faster	(1) Because it enables entirely new solutions	(2) Use algorithmic analysis
(2) Use a completely new software design	(2) Use a GPU	(2) Use Multithreading

Performance

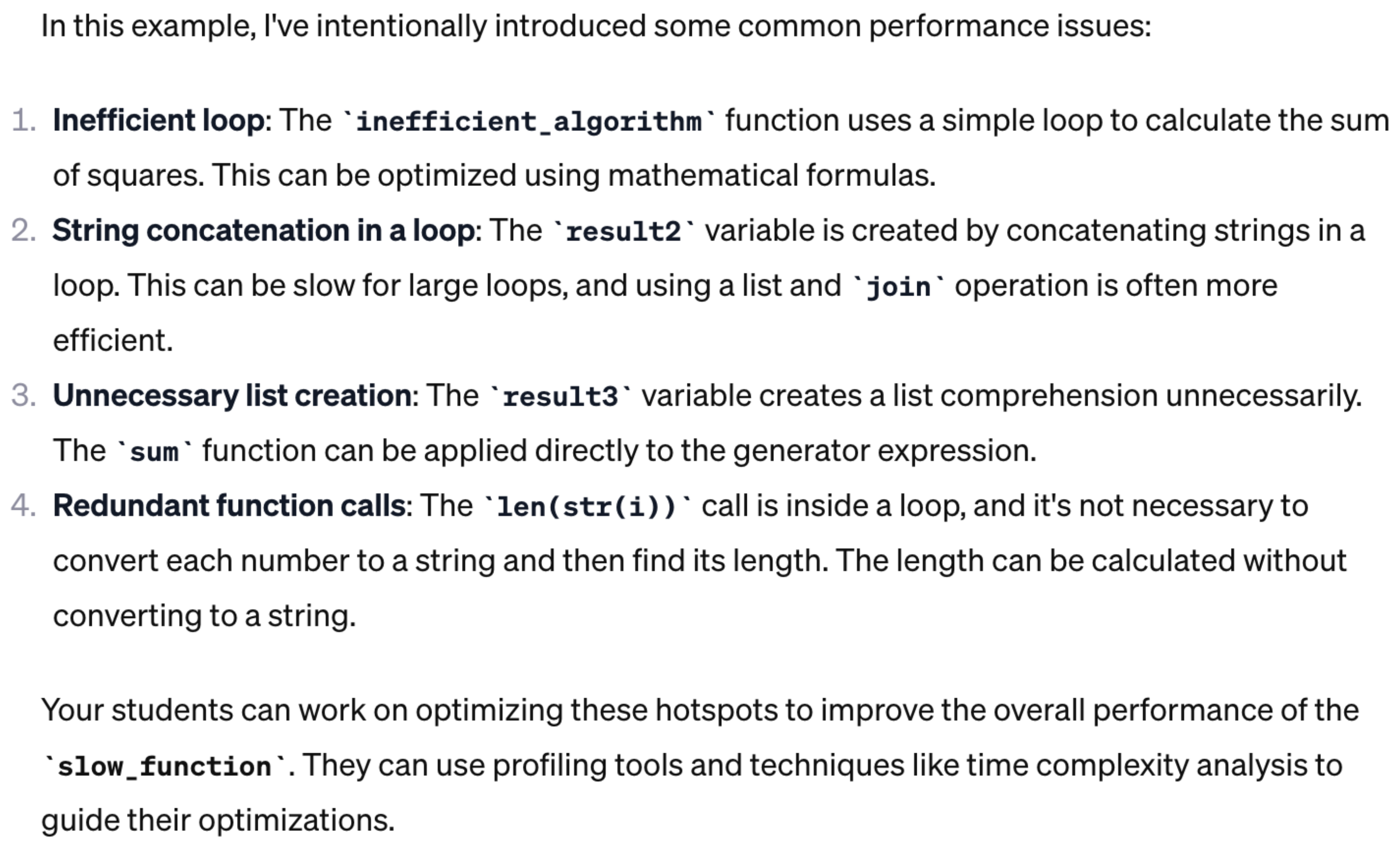
(possible solutions – **tl;dr, it depends on your problem**)

Technique	Potential Effectiveness	Potential Cost
Redesign Software	Medium-High	Very High (depends what you're redesigning)
Change Algorithms	Very Very High	Low, or Very Very Very High (depends on what alg. you need)
Find hotspots in existing code and fix them	**Medium - High**	**Low – Medium** **(depends on the hotspot...)**
Multithreading	Low-Medium	Medium – High (depends on what alg. you're using, and available hardware)
GPUs	Low-High	Medium - Very High (depends on what alg. you're using, and available hardware)
(specialized hardware: FPGAs, ASICs, etc.)	High-Very High (if it suits)	Insanely High (extremely specialist development and hardware required)
Use a "faster" programming language	Medium	Very High (depends on alg, and your software design+size)

Performance
(related quotes)

- “Premature Optimization is the Root of all Evil” - Donald Knuth
- “You can spend a lifetime getting 95 % of your code to be parallel and never achieve better than a 20x speedup – regardless of how many processors you throw at it” - Amdahl’s Law (abridged version)
- **“You can’t make it better until you make it work”** – Akin’s Laws of Spacecraft Design
- “**Engineering is done with numbers. Analysis without numbers is only an opinion**.” - Also Akin’s Laws of Spacecraft Design

(I was too lazy to write the code, but ChatGPT’s insights are effectively exactly what I’d say)



In this example, I've intentionally introduced some common performance issues:

1. **Inefficient loop**: The `` `inefficient_algorithm` `` function uses a simple loop to calculate the sum of squares. This can be optimized using mathematical formulas.
2. **String concatenation in a loop**: The `` `result2` `` variable is created by concatenating strings in a loop. This can be slow for large loops, and using a list and `` `join` `` operation is often more efficient.
3. **Unnecessary list creation**: The `` `result3` `` variable creates a list comprehension unnecessarily. The `` `sum` `` function can be applied directly to the generator expression.
4. **Redundant function calls**: The `` `len(str(i))` `` call is inside a loop, and it's not necessary to convert each number to a string and then find its length. The length can be calculated without converting to a string.

Your students can work on optimizing these hotspots to improve the overall performance of the `` `slow_function` ``. They can use profiling tools and techniques like time complexity analysis to guide their optimizations.

Practical: Find/Fix Hotspots

(the old-fashioned way)

- Go to the meetup’s github
  - https://github.com/adamkewley/rse-meetups
- In this session (**6_Performance**), copy + paste either of these python codes into your IDE and get them running:
  - example.py, for a basic example
  - example_numpy.py, for a numpy-based example
- Use python `time` to record the time taken at certain parts of the script to figure out which parts are running slowly
  - If you’re feeling fancy, you can also look into the `timeit` module
- Is there any way to make them run quicker? (I honestly don’t know – the examples are from ChatGPT ;))

Profilers

- Code profilers are tools that you usually use to run your code (or alongside your code)
- They *typically* (not always) either work by:
  - Sampling: pause your program, record what it was doing when paused, continue
  - Virtualization: run your program in a virtual environment with counters on all instructions
  - Annotations/coloring: parts of your application are automatically annotated with counters etc.
- Python comes with two, `cProfile` (sampling, recommended) and `profile` (annotation-based):
  - https://docs.python.org/3/library/profile.html

Practical #2: Find Hotspots

(but this time, try a profiler)

- Get the example code again
  - Or try `example_longer.py`, if you want some variety
- Run the script via a profiler on the command-line with:
  - python -m cProfile example.py
- Does the output help identify any performance problems?
- Try the same, but with `profile`:
  - python -m profile example.py
- What’s the difference?

Conclusions (performance)

- Top priority is still as-stated in the design lesson: make it simple and make it work
- … But if you think you have a performance issue, then consider the options
- And the practical things we’ve quickly looked at here (measure it, find hotspots, then analyze the hotspots) are generally reliable techniques that work across multiple projects/languages etc.
- But sometimes:
  - “You can't get to the moon by climbing successively taller trees”

Conclusions (overall)

- We covered the basics of software engineering. Hopefully, some of it was useful
- The main take-home is that software is hard to get right. The main thing to optimize for, is moving forward:
  - Maintain a fast iteration loop
  - Keep it simple
  - Know what you’re making
  - Make it work

Future

- Now that the fundamentals are covered, and they don’t really change between languages/projects, future sessions will probably focus on specific/specialized topics
- Or we can have a general “get together and talk about something software-engineering-ey” meetup
- Probably in 2024