

Research Software Engineering Meetups:
“ “.join([“software”.capitalize(),
“implementation”.capitalize()])

2023/11/03

Overview

- `print_session_table(sessions)`
- `introduce(topics.software_implementation)`
- `await attendees.implement_stuff()`

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	In Development
13th November	34-E-3180	(Sub)processes	Proposed
20th November	34-E-3180	(topic TBD: based on feedback)	

source: <https://github.com/adamkewley/rse-meetups/>

topic = “implementation”

- You’ve got the tools
- You’ve got a platform
- You have an idea of what you want to build
- ... Now there’s just the small matter of actually making it

Overview

- It's hard, *especially* if you aren't using the right tools, platform, or design
- You will probably screw it up, many times - regardless of how long you've been doing it
- Your colleagues might make it look easy, but that's because they only show the 5 % that worked during their presentations

- The biggest difference between inexperienced and experienced developers is that the experienced ones tend to anticipate/handle the screw ups faster than the inexperienced ones
- And the experienced ones typically invest time/money into making sure their iteration loop is short (previous session: iteration speed > simplicity > correctness > performance)
- Therefore, get familiar with edge-cases, error messages, common software issues, common gotchas, and figure out which tools help spot/report problems quickly
- Because learning how to work through problems quickly is a much better return-on-investment in software than trying to learn to never make mistakes

Interactive #1

Lets implement sequence reversal

Rules:

- Define a function called `reverse` (i.e. `def reverse(lst):``)
- It should take a single list argument
- It should return a new list such that:
 - `new_list[i] == lst[len(lst)-1-i]` for all `i` in `[0, len(lst))`
- You can't use any library functions, just the python language

Interactive #1

Summary

- Figured it out?
- Did tooling help at all? What tools did you end up using?
 - ✓ Debugger?
 - ✓ REPL?
 - ✓ Wrote a command-line thing in the script that you ran to see the input/output while you worked on it?
- Are you sure it works?
- Are you *really* sure it works?
- Like, really?
- How would you convince me that you're sure?
- How would you convince yourself that you're still sure in a few year's time, after your code has undergone a few hundred changes?

Interactive #2 (shared)

Lets solve a HackerRank puzzle together

- I'll open one up
- We'll give it a whirl
- Hopefully it provides some insights
- (or, at least, it'll show you the wonderful world of HackerRank/LeetCode/etc. for code interviews)

Interactive #2

Summary

- What's nice about this (way of approaching coding)?
- What's hard about this?
- Is there something we can learn about these systems?
- Is it a useful study guide/teacher?

Testing

- Testing can elicit feelings of intimidation and boredom at the same time ;)
- But you're probably already doing it, it's just that you haven't been calling it testing

- If you have a habit of regularly running your program and checking outputs (you should), you're testing
- The only thing you're missing is to *automate* and *standardize* the testing process

Sidenote: assertions

- Assertions are runtime tests that you place in your code that explode (i.e. throw an exception) if the statement in them is `False`
- Python has in-built support for assertions:

```
assert len(lst) > 0
```

- **You should use them in your code.**
 - If your code assumes something (e.g. “the average of this numpy vector is non-negative”)
 - And the performance overhead of double-checking isn’t important
 - Then an assertion is MUCH better than a comment – comments aren’t checked, assertions are
 - And if performance becomes a problem, then they’re easy to find+remove later
- Automated testing is effectively the act of running external suites of assertions against your code

Interactive #3

Add automated tests for `reverse`

- Use the python `unittest` module (comes with python) to write some basic unit tests for your `reverse` function
- The test should explode/fail when something doesn't meet expectations. You can use `assert (expr)` to do this. `unittest` also provides useful assertions (`self.assertEqual`, etc.)
- Initially, you can put your `unittest.TestCase` in the same file, and then call `unittest.main()` to run all of the tests
- Later, you can move the test cases into separate python modules and run them separately.

Interactive #3

Summary

- Is it easy enough to add some tests?
- How rigorously should you test?
- Should tests be committed? Or just used as a temporary implementation aid?


Interactive #4 (bonus)

Lets implement a (binary) search algorithm

Rules:

- Define a function called ``binary_search`` (i.e. ``def binary_search(lst, el):``)
- It should take a single list argument (``lst``) and an element to search (``el``)
- Elements in ``lst`` will be provided sorted, and each element will be comparable (`<`, `>`, `==`) to ``el``
- Your function should return the index of ``el`` in ``lst``
- Input size: $0 \leq \text{len}(\text{lst}) < 10^8$
- Run time must be “fast” (i.e. < 0.1 seconds, even for large inputs)
- You can't use any library functions, just the python language

Next Time (up for debate)

 **Note:** This list is just to give you an idea (it's very very provisional).

Topic	Description
Tools	IDEs, text editors, REPL, command line, basic git usage, LLMs (ChatGPT/GH Copilot)
Sharing, Publication, and Collaboration	Shared drives, GitHub, GitLab, writing a README, managing dependencies, releasing, publishing to package managers like <code>pip</code> , etc.
Software Design	Library vs. command-line vs. UI. User-/developer-experience. What is an application? The basics of how applications work.
Software Implementation	Iterative development methods. The REPL. Incremental application design. Assertions. Tests. How to implement things done more quickly - and with fewer surprises.
Libraries	Command-line parsers, configuration parsers, plotters, renderers
Advanced/Specialized topics (later)	Languages with type systems. Native application development (C/C++/Fortran /Rust). Multithreading. GP-GPU. Julia, hardware engineering, etc.