~~Best~~ Good Practices for Research Software Engineering

Ian A. Cosden, Ph.D.
Manager, HPC Software Engineering and Performance Tuning
Research Computing, Princeton University

Background

- Much science would be impossible without research software
- “Research software”
  - Anything used to generate, process, or analyze results you intend to appear in a publication
  - Anything from a few lines of code written by you to a professionally developed software package
- Hardware comes and goes but [good] software is forever
- Scientists usually develop their own software
  - Spend a lot of time writing it
  - Most (90%)[1] are self-taught
  - Don’t know what “good” software even looks like

1. Hannay JE, et al.. (2009) “How do scientists develop and use scientific software?” Proceedings Second International Workshop on Software Engineering for Computational Science and Engineering.

Motivation

- Many domain scientists think programming is a necessary tax/evil
- Many don’t care or think about reproducibility or sustainability…
- …But everyone wants to be more productive

> “Good programmers are 10x more productive than average.”
>
> -- Software engineering folklore

Mission

- Present a handful of practices & guidelines to make you more productive
  - Are time tested
  - Low technical barrier to entry
  - Anyone can adopt, including you
- These are <u>not</u> rules
- Not necessarily perfect or best, but good or “*good enough*”

Business image created by Jigsawstocker - freepik.com

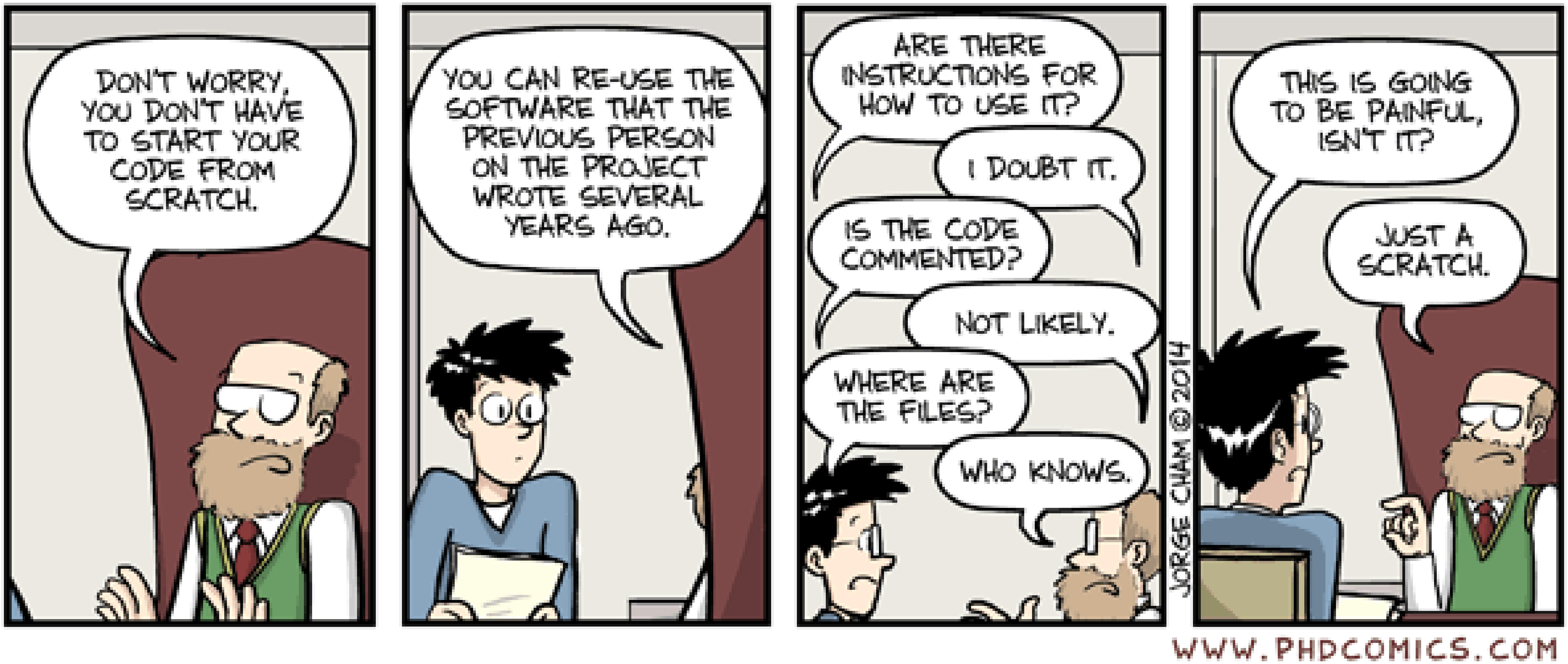
Acknowledgment & References

A significant portion of this work isn’t new. It’s been inspired by and borrowed from:

1. Wilson, G. et al. Good Enough Practices for Scientific Computing. arXiv:1609.00037. 2016.
2. Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, et al. (2014) Best Practices for Scientific Computing. PLoS Biol 12(1): e1001745. https://doi.org/10.1371/journal.pbio.1001745
3. Greg Wilson: "Software Carpentry: Lessons Learned". F1000Research, 2016, 3:62 (doi: 10.12688/f1000research.3-62.v2).
4. Software Carpentry: http://software-carpentry.org/ (licensed by CC BY 4.0)

Don’t be the “previous person!”

Piled Higher and Deeper ***by Jorge Cham***



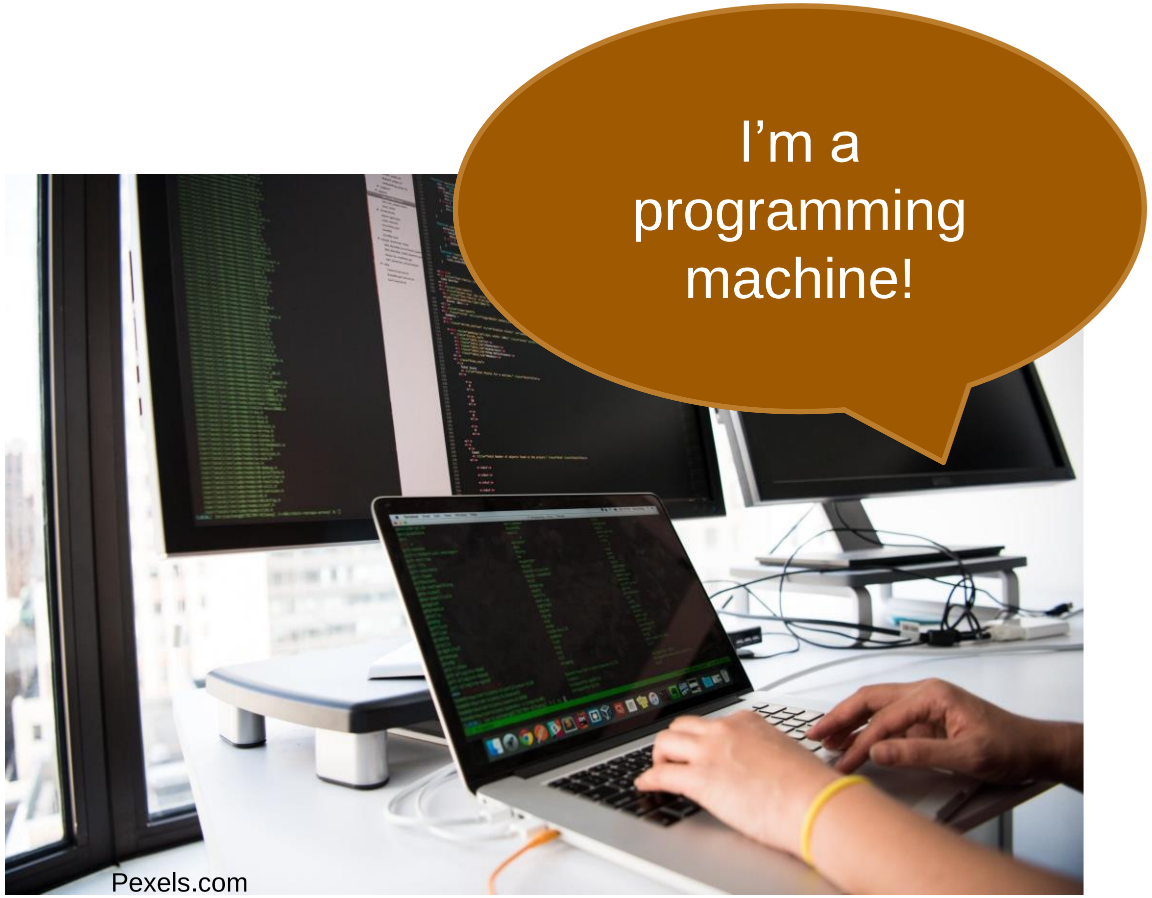
http://phdcomics.com/comics.php?f=1689

1. Write programs for people, not computers

- Code that is difficult to understand is:
    - Hard to tell if it’s doing what it's supposed to
    - Hard for others to re-use it…
    - ...including **your future self**

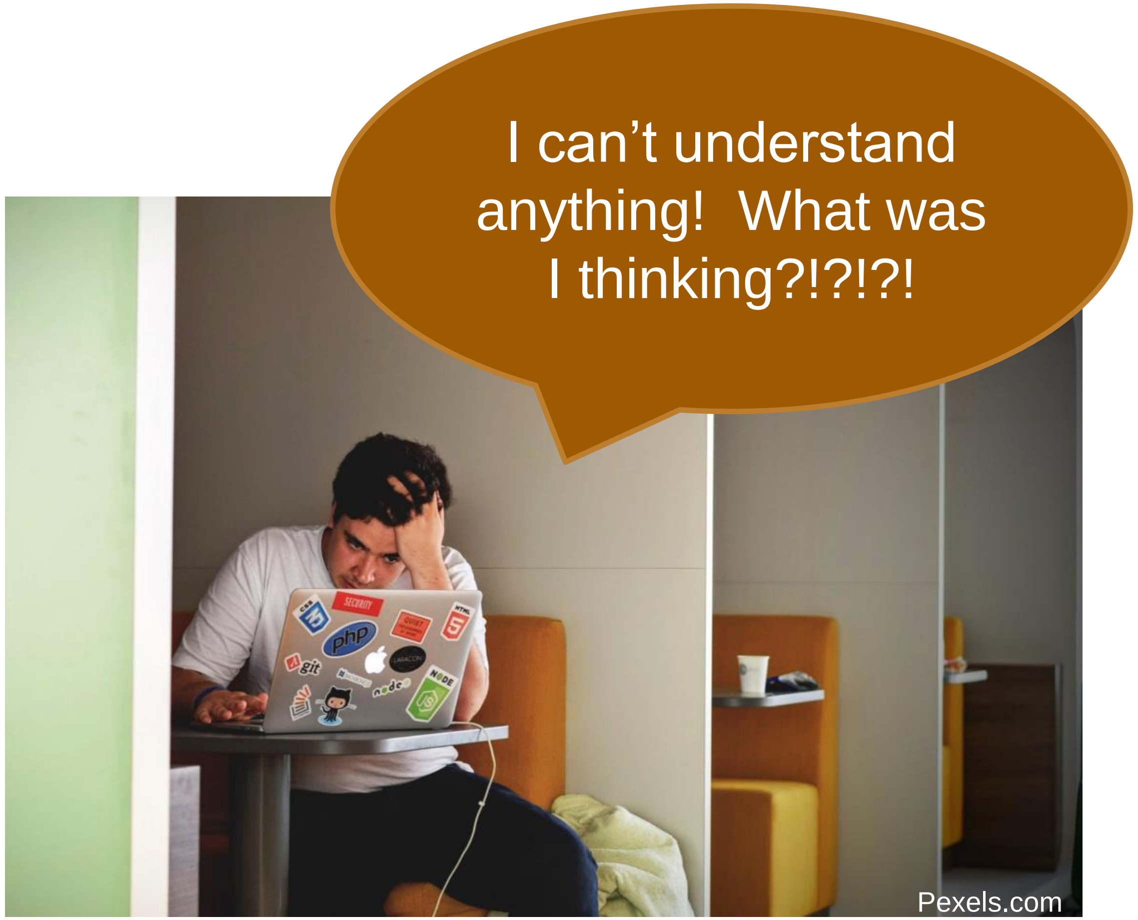
1. Write programs for people, not computers

Today



Pexels.com

1 year from now



Pexels.com

1. Write programs for people, not computers

- Goal should be to make the next person's life easier
- Give a usage example
- Focus on things the code doesn't say or doesn't say clearly

Worthless comment:

```
i = i + 1    # increment i by 1
```

1a. Make names consistent, distinctive, and meaningful

- `p` doesn't help the reader as much as `pressure`
- Don't use `temp` for both "temporary" and "temperature"
- `i`, `j` are OK for indices in small scopes
- Do it well: self-documenting code

Don’t do this:

```
subroutine seg_lj_efo_ist(lmpvec,cmovec,ij_1,lam_all,virt_nu_all, ijab_sq, &
                      Tx,Ty,scep_C_int,scep_EI_int,scep_CL_1,scep_alpha_1, &
                      xi_data,realI,realJ)
```

1a. Make names consistent, distinctive, and meaningful

- It’s harder, but worth the effort

Internet meme – origin unknown (reddit.com)

1b. Make code style and formatting consistent

- Which rules don't matter — having rules does
- Brain assumes all differences are significant
- Every inconsistency slows comprehension

```
int molecule_flag;
int q_flag mu_flag;

int rmass_Flag,RadiusFlag,OmegaFlag,Torque_Flag,AngmomFlag;
```

1c. Document interfaces and reasons not implementations

- Interfaces and reasons change more slowly than implementation details
- Place a brief explanatory comment at the start of every program

Useful[1]:

```
Synthesize image files for testing circularity estimation algorithm.

Usage: make_images.py -f fuzzing -n flaws -o output -s seed -v -w size

where:
-f fuzzing = fuzzing range of blobs (typically 0.0-0.2)
-n flaws   = p(success) for geometric distribution of # flaws/sample (e.g. 0.5-0.8)
-o output  = name of output file
-s seed    = random number generator seed (large integer)
-v         = verbose
-w size    = image width/height in pixels (typically 480-800)
-h         = show help message
```

1. Wilson, G. et al. Good Enough Practices for Scientific Computing. arXiv:1609.00037. 2016.

1d. Break programs into short, readable functions

- Short-term memory can hold 7±2 items
- So break programs into short, readable functions, each taking only a few parameters

Even more reason not to do this:

```
subroutine seg_lj_efo_ist(lmpvec,cmovec,ij_1,lam_all,virt_nu_all, &
                ijab_sq,Tx,Ty,scep_C_int,scep_EI_int,scep_CL_1,scep_alpha_1, &
                xi_data,realI,realJ)
```

Practice 1 – Another example

```
# calculate rectangle area
def calc_rect_area(x1, y1, x2, y2):
    calculation...
```

```
sa = calc_rect_area(x1, y1, x2, y2)
```

```
sa = calc_rect_area(x1, x2, y1, y2)
```

```
# given two points
#       point1(x1, y1)
#       point2(x2, y2)
# returns area of resulting rectangle
# If points do not form rectangle, returns -1

def calc_rect_area(point1, point2):
    calculation...
```

```
surface_area = calc_rect_area(point1, point2)
```

```
surface_area = calc_rect_area(point2, point1)
```

Practice 2 – Let computers do the work

- Computers exist to repeat things quickly
- If you are 99% accurate
    - 63% chance that at least one error per 100 repetitions

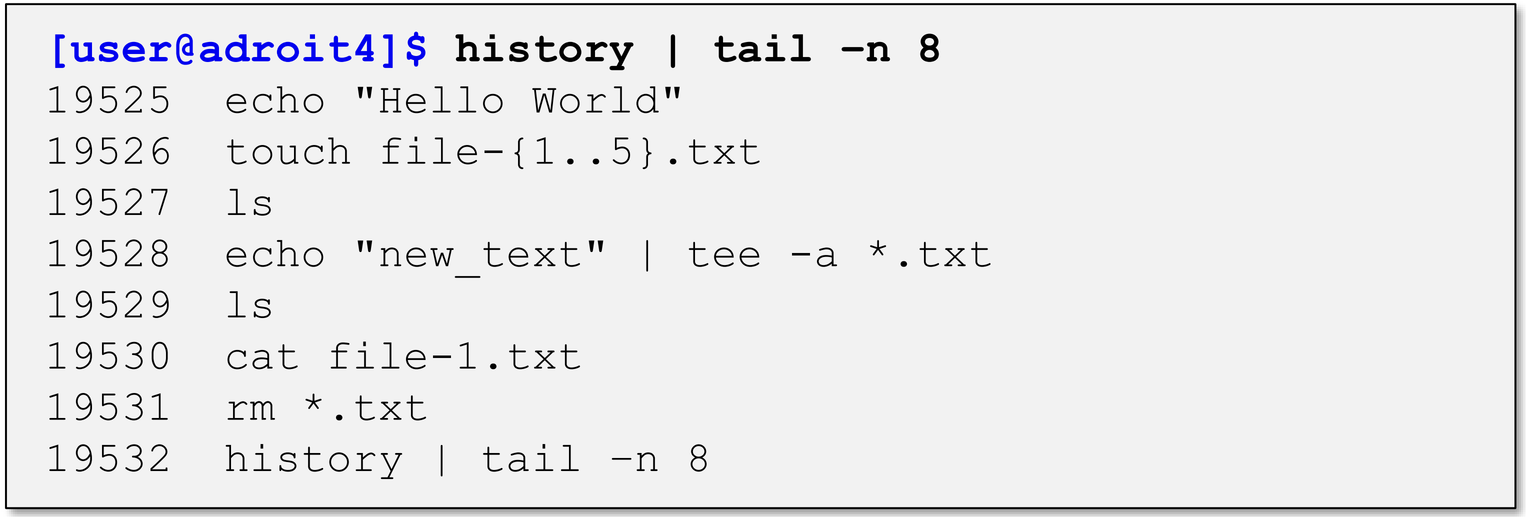
2a. Make the computer repeat tasks

- Write little programs for everything
- They can be called scripts, macros, or aliases
- Easier to do this with text-based programming systems than with a GUI
  - Command line!

```
[icosden@adroit4 log_file_scripts]$ ls
convert_dates.sh
print_unique_users.sh
print_users_by_month.sh
process_audit_log.py
```

2b. Save recent commands in a file for re-use

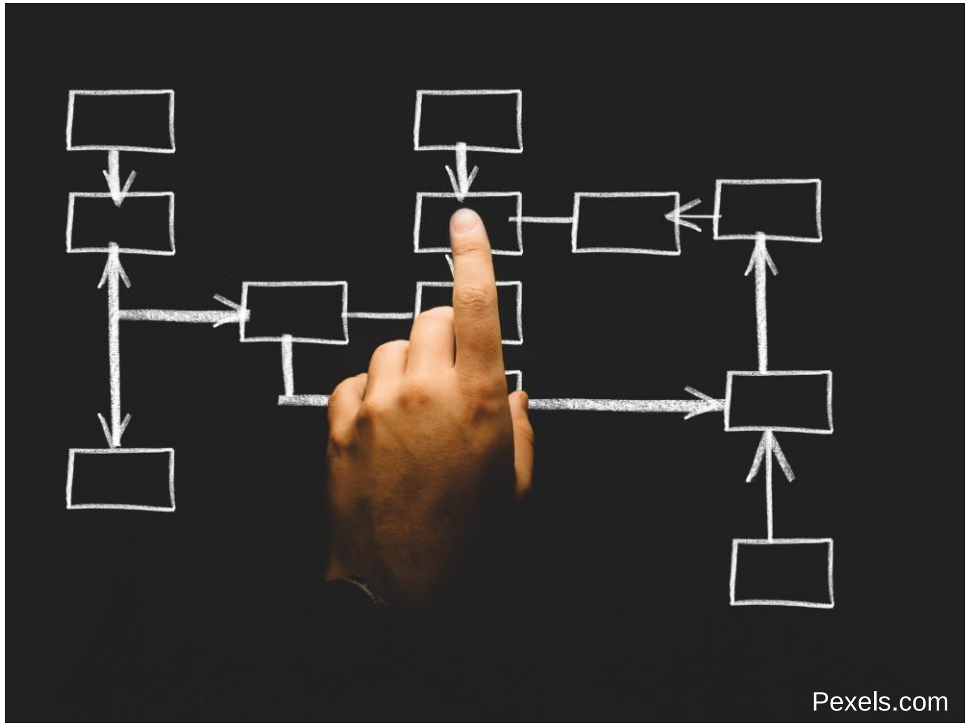
- Most text-based interfaces do this automatically
    - Repeat recent operations using history
    - "Reproducibility in the small"
- Save your history in a file or use your history to build a script
    - An accurate record of how a result was produced
    - Only if everything can be captured



```
[user@adroit4]$ history | tail -n 8
19525  echo "Hello World"
19526  touch file-{1..5}.txt
19527  ls
19528  echo "new_text" | tee -a *.txt
19529  ls
19530  cat file-1.txt
19531  rm *.txt
19532  history | tail -n 8
```

2c. Automate workflows

- Originally developed for compiling programs
- Can be used whenever some files depend on others
- Makes workflow explicit


Pexels.com

Practice 2 Example - Makefile

```
.PHONY: all hello_world generate_files
populate_files clean

all: hello_world generate_files populate_files


hello_world:
        @echo "Hello World!"

generate_files:
        @echo "Creating empty text files..."
        touch file-{1..5}.txt

populate_files:
        @echo "Adding text to each file"
        @echo "new_text" | tee -a *.txt > /dev/null

clean:
        @echo "Cleaning up..."
        rm *.txt
```

```
[user@adroit4]$ ls
Makefile
[user@adroit4]$ make
Hello World!
Creating empty text files...
touch file-{1..5}.txt
Adding text to each file...
[user@adroit4]$ ls
file-1.txt  file-2.txt  file-3.txt
file-4.txt  file-5.txt  Makefile
[user@adroit4]$ make clean
Cleaning up...
rm *.txt
```

Makefiles are great for repeatably building/compiling code

Practice 3 – Make incremental changes

Pexels.com

3a. Use a version control system

- Track changes
- Allows them to be undone
- Supports independent parallel development
- Essential for collaboration
    - Please, no more emailing code

git

SUBVERSION

mercurial

GitHub

GitLab

Bitbucket

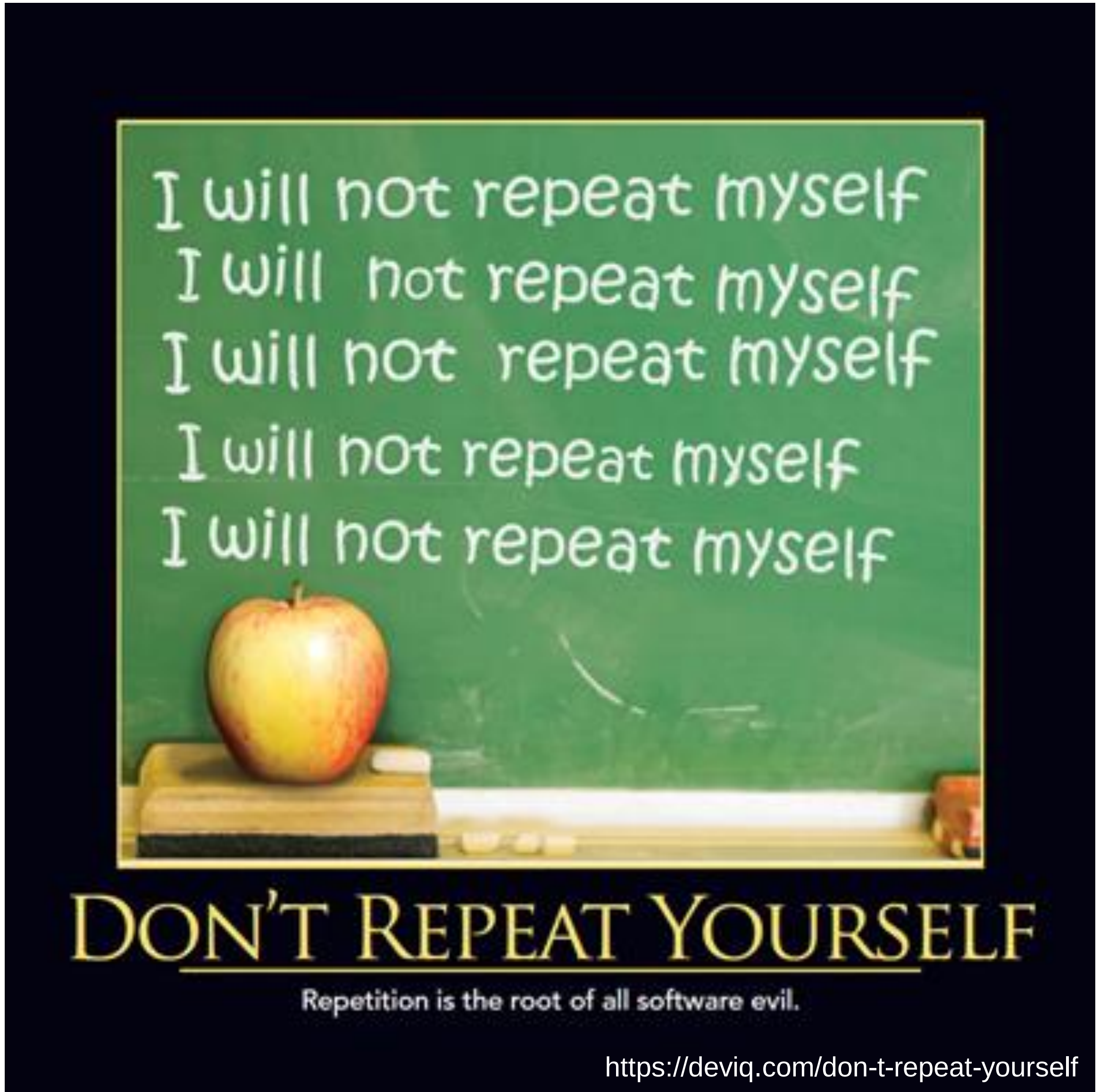
3b. Put everything that has been created manually in version control

- Not just software
    - Papers, raw images
    - Not gigabytes, but *metadata* about those gigabytes
- Leave out things generated by the computer
    - Use build tools to reproduce those instead
    - Unless they take a very, very long time to create
- Back up (almost) everything created by a human as soon as it is created

3c. Work in small steps with frequent feedback and course correction

- Break larger tasks into small realistic steps
    - 1-2 hours chunks ideal
- Goal: produce (incomplete) working code

Practice 4 – Don’t repeat yourself (or others)



https://deviq.com/don-t-repeat-yourself

4a. Every piece of data must have a single authoritative representation in the system

- Be ruthless about eliminating duplication
- Define constants exactly once

File 1

```
omp_set_num_threads(4)
... calculations ...
... calculations ...

omp_set_num_threads(4)
... calculations ...
... calculations ...
```

File 2

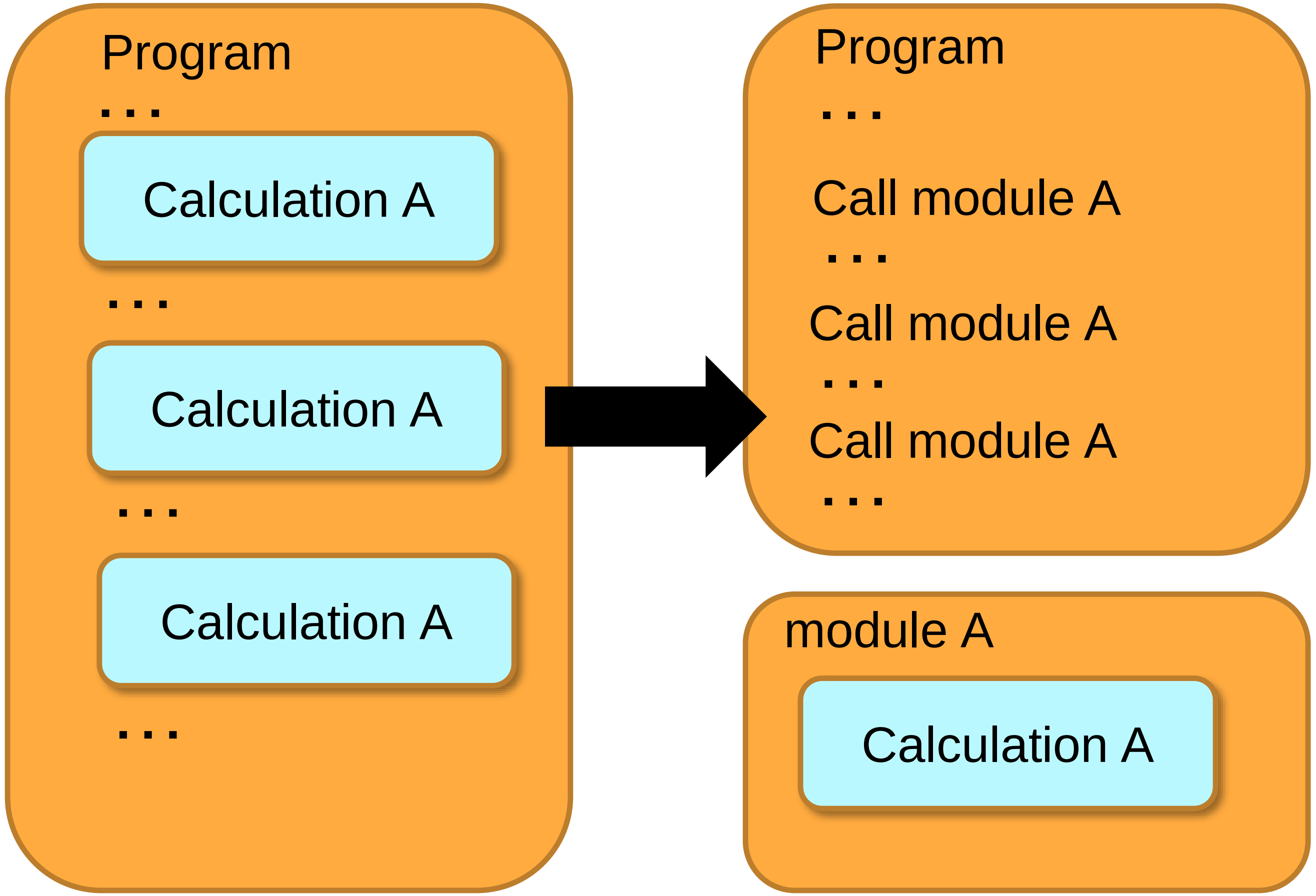
```
omp_set_num_threads(4)
... calculations ...
... calculations ...
```

File 3

```
omp_set_num_threads(4)
... calculations ...
... calculations ...

omp_set_num_threads(4)
... calculations ...
... calculations ...
```

4b. Modularize code rather than copying and pasting

- Reducing code cloning reduces error rates
- Cuts the amount of testing needed
- Increases comprehension



4c. Re-use code instead of rewriting it

- Search for *well-maintained* software libraries & packages
- It takes experts years to build high-quality numerical or statistical software
- Your time is better spent doing science on top of that

PETSc

OpenBLAS

NumPy

Practice 5 – Plan for mistakes

- No single practice catches everything
- So practice defense
- Improving quality increases productivity

```
assert(phone.alive() == true);
```

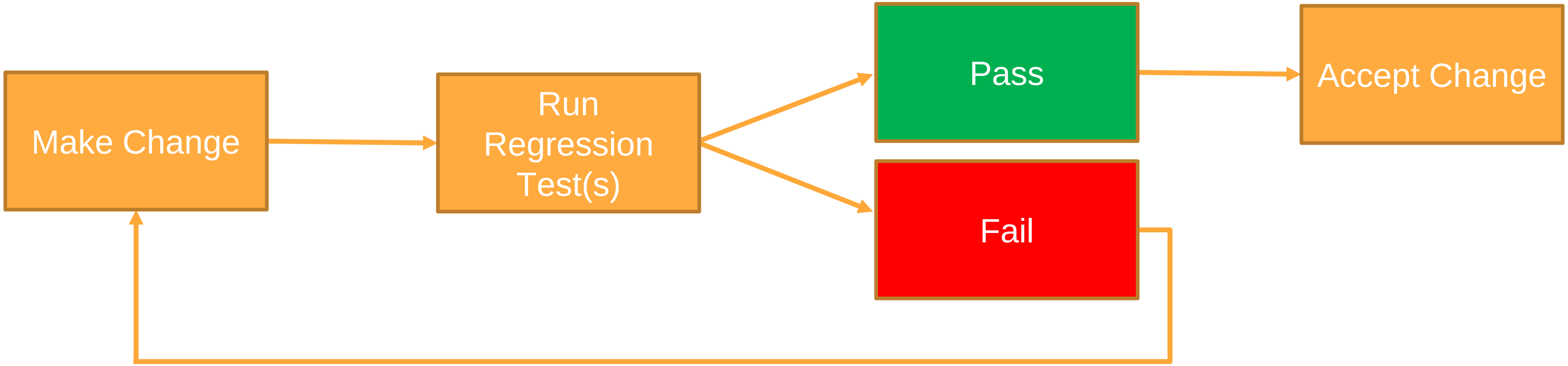
5a. Add assertions to programs to check their operation

- "This must be true here or there is an error"
- No point proceeding if the program is broken or will give wrong results
- Serves as executable documentation

```
void MoveParticles(const int nParticles, ParticleArrays &particle, const float dt) {

        const int tileSize = 128;

        assert(nParticles%tileSize == 0);

        ...
}
```

5b. Set up tests

- Regression Tests
  - Script(s) that check for correctness
  - Run after every change to the code
- Testing is hard!
  - “Correct” answer isn’t always easy
  - Forced documentation as to what is acceptable



5c. Use a symbolic debugger

- Explore the program as it runs
- Better than print statements
    - You don't have to guess in advance what you'll need to know
    - Or re-run every time
- Use breakpoints to:
    - Stop program at a particular points
    - When particular things are true

arm DDT

PC PyCharm

RStudio®

MATLAB®

TotalView®

ATOM

Practice 6 – Optimize software only after it works correctly

- Even experts find it hard to predict performance bottlenecks
- Small changes to code can have dramatic impact on performance
- What if you end up never using that code?
- Get it right, *then* make it fast

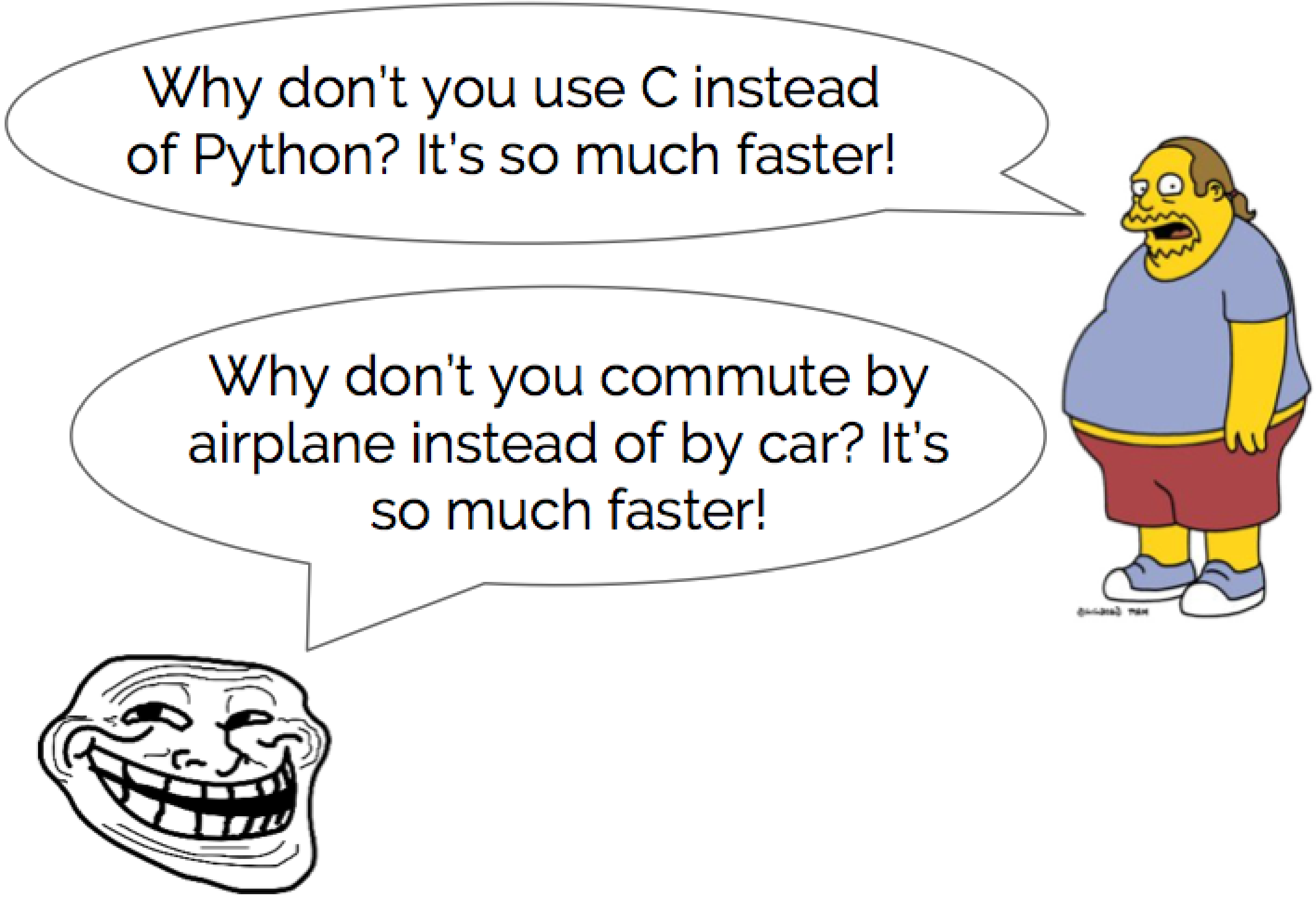
> “Premature optimization is the root of all evil.”
>
> *-- Donald Knuth, Professor Emeritus of The Art of Computer Programming at Stanford University*

Pexels.com

6a. Write code in the highest-level language possible

- For day-to-day scientific data exploration
    - Speed of development is primary
    - Speed of execution is often secondary
- So use the most expressive language available to get the "right" version
- Then rewrite core pieces (possibly in a lower-level language) to get the "fast" version

6a. Write code in the highest-level language possible



6b. Use a profiler to identify hotspots

- Reports how much time is spent in each function and on each line of code
- Re-check on new hardware or when switching libraries
- Don’t guess or assume

arm
MAP

Python cProfile

MATLAB

RStudio

NVIDIA Visual Profiler

Summary: The high level practices

1. Write programs for people, not computers
2. Let the computer do the work
3. Make incremental changes
4. Don’t repeat yourself (or others)
5. Plan for mistakes
6. Optimize software only after it works correctly

Additional topics

- More testing
    - Continuous Integration
    - Unit tests
    - Test Driven Development (TDD)
        - “Red, Green, Refactor”
- Documentation tools
    - Sphinx, Doxygen
- Issue tracking tools

- Code management
    - Logical organization of project files
- Data management
    - Saving raw and intermediate forms
    - Documentation
- Collaboration
    - Pull requests (merge requests)
    - Code review
    - License: make it explicit
    - Publish: make it citable

Final Remarks

- Remember, these are good practices not rules
- Think: make conscious decisions
- Invest: take the time to do it right