# Physics 239
# Radiative Processes in Astrophysics

Lecture #3: Scientific Computing
then Two-Level Atom, scattering & more
radiative transfer

# Scientific Computing

based on Wilson et al. 2014, PLoS Biology, Vol 12, Issue 1
"Best Practices for Scientific Computing"
and slides from Software Carpentry: http://swcarpentry.github.io/slideshows/best-practices/

Many of us get no formal training in scientific computing.
We pick it up as we go along.

However, it is a large part of what we do.

I recommend reading the above paper and trying to implement its suggestions in your assignments for this class.

# Summary of Best Practices

- Write programs for people not computers.
- Let the computer do the work.
- Make incremental changes.
- Don’t repeat yourself (or others).
- Plan for mistakes.
- Optimize software only after it works correctly.
- Document design and purpose, not mechanics.
- Collaborate.

# Write programs for people not computers.

People should be able to read your code and comments and understand what the program does.

People includes *future you* who will forget what you did.

Reproducibility is a goal for all of your scientific efforts.

# Write programs for people not computers.

- A program should not require its readers to hold more than a handful of facts in memory at once.
    - Break programs into short, readable functions taking only a few parameters.
- Make names consistent, distinctive, and meaningful.
- Make code style and formatting consistent.

# Let the computer do the work.

- Make the computer repeat tasks.
- Save recent commands in a file for re-use.
- Use a build tool to automate workflow.

Script your work.

# Let the computer do the work.

In order to maximize reproducibility, everything needed to re-create the output should be recorded automatically in a format that other programs can read. (Borrowing a term from archaeology and forensics, this is often called the *provenance* of data.) There have been some initiatives to automate the collection of this information, and standardize its format [43], but it is already possible to record the following without additional tools:

- unique identifiers and version numbers for raw data records (which scientists may need to create themselves);
- unique identifiers and version numbers for programs and libraries;
- the values of parameters used to generate any given output; and
- the names and version numbers of programs (however small) used to generate those outputs.

Wilson et al. 2014

# Make incremental changes.

> Unlike traditional commercial software developers, but very much like developers in open source projects or startups, scientific programmers usually don't get their requirements from customers, and their requirements are rarely frozen [31,44]. In fact, scientists often *can't* know what their programs should do next until the current version has produced some results. This challenges design approaches that rely on specifying requirements in advance.

- Work in small steps with frequent feedback and course correction.
- Use a version control system.
- Put everything that has been created manually in version control.

# Don’t repeat yourself (or others).

“Anything repeated in two or more places will eventually be wrong in at least one.”

- Every piece of data must have a single authoritative representation in the system.
    - Example: defining constants once rather than in each program.
- Modularize code rather than copying and pasting.
- Re-use code rather than rewriting it (e.g. astropy or IDLAstro lib).

# Plan for mistakes.

"defensive programming"

- Add "assertions" to programs to check their operation. (to me this is: add error checking and output feedback throughout program)
- Unit testing.  Re-run after any changes to the code.
- Turn bugs into test cases.
- Use an "interactive program inspector" for debugging. Use breakpoints to stop program at specific points.

# Optimize software only after it works correctly.

"get it right, then make it fast"

- Use a "profiler" to identify bottlenecks.
    - reports how much time is spent on each part of code
- Write code in the highest level language possible.
    - people write ~constant number of lines of code per hour regardless of language, use the high level language first, then optimize for speed in lower level language if needed

# Document design and purpose not mechanics.

- Document interfaces and reasons, not implementations.

- Embed the documentation for a piece of software in that software

# Collaborate

- Get collaborators to review code.
- Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems.
- Use an issue tracking tool.

# Key Points for Physics 239

- Script your code.
- Version control.
- Write tests to make sure your code works the way you want.
- Document your code.