Software Engineering Practices for Python

Coding Experiences

- Good development practices help with the following situations:
  - *You swear that the code worked perfectly 6 months ago*, but today it doesn't, and you can't figure out what changed
  - *Your research group is all working on the same code*, and you need to sync up with everyone's changes, and make sure no one breaks the code
  - *Your code always worked fine on machine X*, but now you switch to a new system/architecture, and you code gives errors, crashes, ...
  - *Your code ties together lots of code*: legacy code from your advisor's advisor, new stuff you wrote, all tied together by a driver. The code is giving funny behavior sometime—how do you go about debugging such a beast?

Software Engineering Practices

- We'll look at some basic python style guidelines and some tools that help with the development process
  - Also helps reproducibility of your science results
  - You can google around for specific details, more in-depth tutorials, etc.

OUR GOAL IS TO WRITE BUG-FREE SOFTWARE. I'LL PAY A TEN-DOLLAR BONUS FOR EVERY BUG YOU FIND AND FIX.
YAHOO!
WE'RE RICH
YES!!! YES!!! YES!!!
I HOPE THIS DRIVES THE RIGHT BEHAVIOR.
I'M GONNA WRITE ME A NEW MINIVAN THIS AFTERNOON!
S. Adams E-mail: SCOTTADAMS@AOL.COM
© 1995 United Feature Syndicate, Inc.(NYC)

Software Engineering Practices

- Some basic practices that can *greatly* enhance your ability to write maintainable code
  - Version control
  - Documentation
  - Testing procedures
  - For compiled languages, I would add Makefiles, profilers, code analysis tools (like valgrind)
- Already discussed: PEP 8 (coding standards)
  - Helps you interact with a distributed group of developers—everyone writes code with the same convention

Python Style

- The recommended python style is described in a “Python Enhancement Proposal”, PEP-8
  - http://legacy.python.org/dev/peps/pep-0008/
  - Based on the idea that “*code is read much more often than it is written*”
- Some highlights:
  - Indentation should use 4 spaces (no tabs)
  - Lines should be less than 79 characters
    - Continuation via '\' or ()
  - Classes should be capitalized of form MyClass
  - Function names, objects, variables, should be lower case, with _ separating words in the name
  - Constants in ALL_CAPS

Coding Style

- Don't make assumptions
  - For if clauses, have a default block (else) to catch conditions outside of what you may have expected
  - Use try/except to catch errors
- Use functions/subroutines for repetitive tasks
  - Check return values for errors
  - Use well-tested libraries instead of rolling your own when possible

Version Control

- Old days: create a tar file with the current source, mail it around, manually merge different people's changes…
- Version control systems keep track of the history of changes to source code
  - Logs describe the changes to each file over time
  - Allow you to request the source as it was at any time in the past
  - Multiple developers can share and synchronize changes
    - Merges changes by different developers to the same file
    - Provide mechanisms to resolve conflicting changes to files
  - Provide mechanisms to create a branch to develop new features and then merge it back into the main source.

Version Control

- Even for a single developer version control is a great asset
  - Common task: you notice that your code is giving different answers/behavior than you've seen in the past
    - Check out an old copy where you know it was working
    - Bisect the history between the working and broken dates to pin down the change
- Can also use it for papers and proposals—all the authors can work on the same LaTeX source and share chages
- All of these slides are stored in version control—let's me work on them from anywhere easily

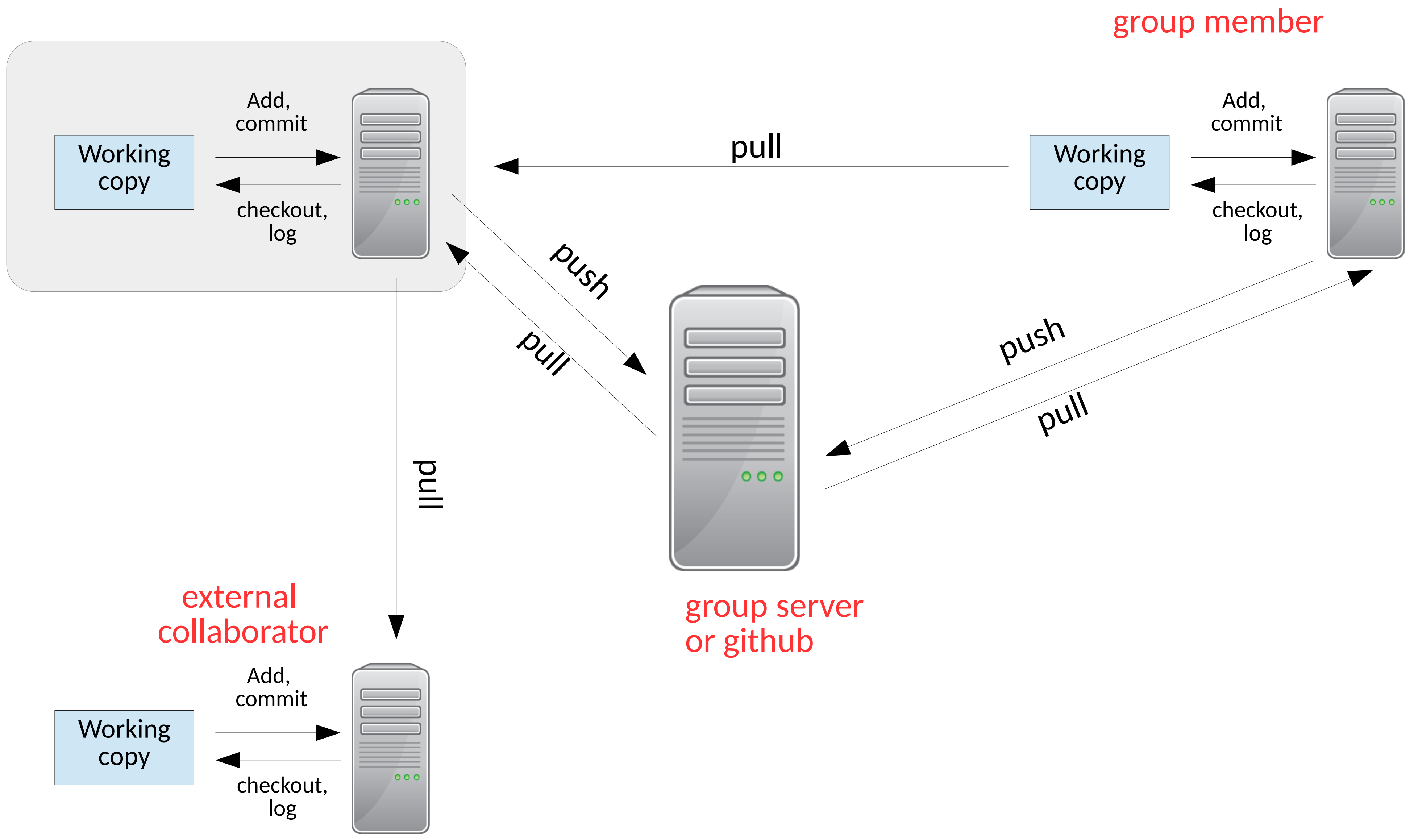
Centralized vs. Distributed Version Control

- Centralized (e.g. CVS, subversion)
  - Server holds the master copy of the source, stores history, changes
  - User communicates with server
    - Checkout source
    - Commit changes back to the source
    - Request the log (history) of a file from the server
    - Diff your local version with the version in the server
  - Doesn't scale well for very large projects
  - "Older" style of version control

- Distributed (e.g. git, mercurial)
  - Everyone has a full-fledged repository
  - You clone another person's repo
  - Commits, history, diff, logs are all local operations (these operations are faster)
  - You push your changes back to others.
  - Each copy is a backup of the whole history of the project
  - Easier to fork—just clone and go

Any version control system is better than none!

Distributed Version Control



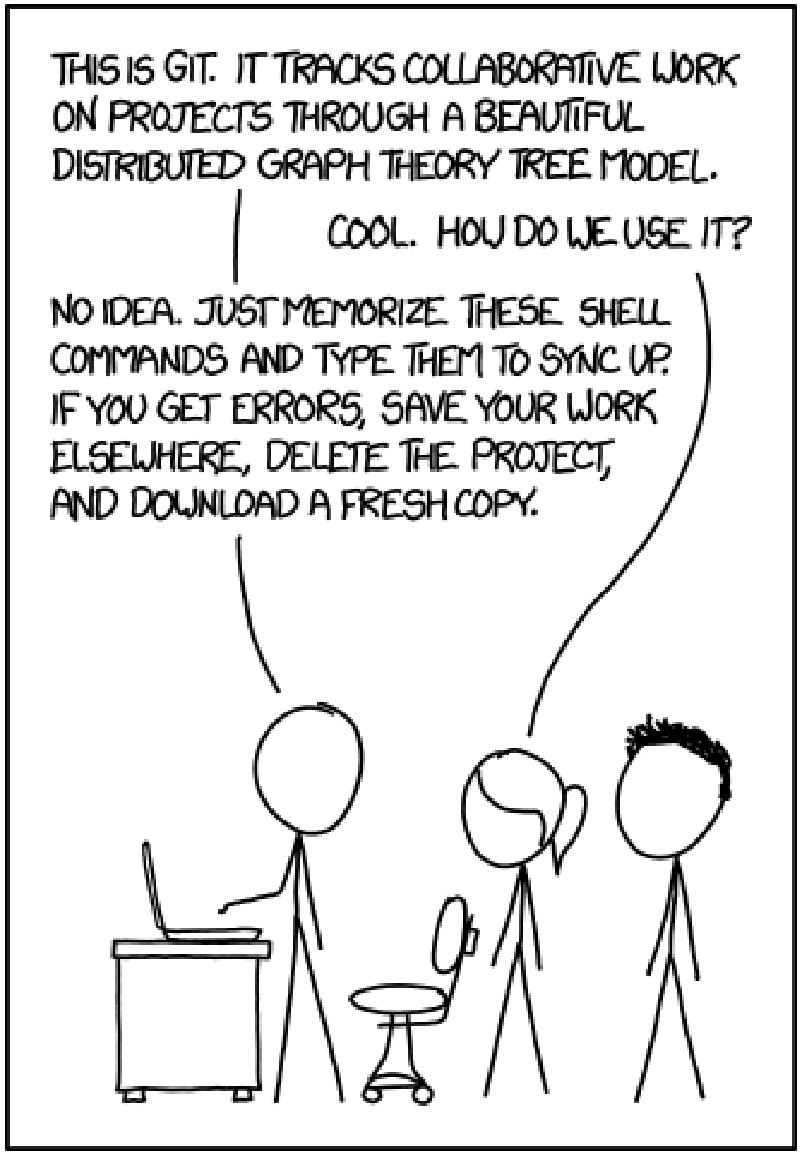
Distributed Version Control

- In an ideal world, people only pull from others, never push.
  - See, e.g. http://bitflop.com/document/111
- Github/bitbucket provide a centralized repo built around *pull requests*

Version Control

- Note that with git, every change generates a new “hash” that identifies the entire collection of source.
  - You cannot update just a single sub-directory—it's all or nothing.
- Branches in a repo allow you to work on changes in a separate are from the main source.
  - You can perfect them, then merge back to the main branch, and then push back to the remote.
- LOTS of resources on the web.
- Best way to learn is to practice.
- There is more than one way to do most things
- Free (for open source), online, web-based hosting sites exist (e.g. Github, BitBucket, …)

Git


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.
(xkcd)

Some git Examples

- We can imagine a many different workflows, like:
  - Single user interacting with a git repo on a single machine
    - This can be, for example, a code that only you develop
    - You want to take advantage of logging/history as well as branching
  - Research group developing a single code base, with centralized server
    - Each of you has a clone of the group's repo and push to/pull from the centralized server to share work
  - Research group using github to centralize the development
    - This is like the previous example, but github fills in for the group's server
    - Built around the idea of pull requests
- We'll look at the first and last of these now

Quick git Example I

- Working with your own repository
- Here's a script to follow along with:
  - https://github.com/marivifs-teaching/python-class/blob/master/lectures/03-practices/git-single.ipynb
  - We'll look at the commands:
    - git init
    - git add
    - git commit
    - git log
    - git checkout
    - git merge

Community

- Github / bitbucket provide tools to engage with your community
- Issue tracking
- Pull requests

LATEST: 10.17
UPDATE
CHANGES IN VERSION 10.17:
THE CPU NO LONGER OVERHEATS WHEN YOU HOLD DOWN SPACEBAR.
COMMENTS:
LONGTIME USER4 WRITES:
THIS UPDATE BROKE MY WORKFLOW! MY CONTROL KEY IS HARD TO REACH, SO I HOLD SPACEBAR INSTEAD, AND I CONFIGURED EMACS TO INTERPRET A RAPID TEMPERATURE RISE AS "CONTROL".
ADMIN WRITES:
THAT'S HORRIFYING.
LONGTIMEUSER4 WRITES:
LOOK, MY SETUP WORKS FOR ME. JUST ADD AN OPTION TO REENABLE SPACEBAR HEATING.
EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

(xkcd)

Github example

- Don't want to use your own server, use github or bitbucket
  - Free for public (open source) projects
  - Pay for private projects
- Create a github account (free)
- These class notes are on github:
  - git clone https://github.com/sbu-python-class/python-science
- Github is great for managing a community of developers outside your organization
  - You don’t have to give everyone write permission
  - Normal interaction is through *pull requests* and *issues*

Github example

- Our class test repo: https://github.com/sbu-python-class/test-repo-2018
  - *Fork* the project into your own account
  - Use git to clone your fork and interact with it—*you own it, so you can push changes back to your fork*
  - Issue a *pull-request* to the main (upstream) project asking for your changes to be incorporated
  - Feel free to try this workflow out with the class repo!

Version Control

- There's no reason not to use version control
  - Pick one (git or hg) and use it
- Even if you are working alone
  - Each clone has all the history of the project
  - Cloning on different machines means you have backups
  - Allows you to sync up your work between home and the office

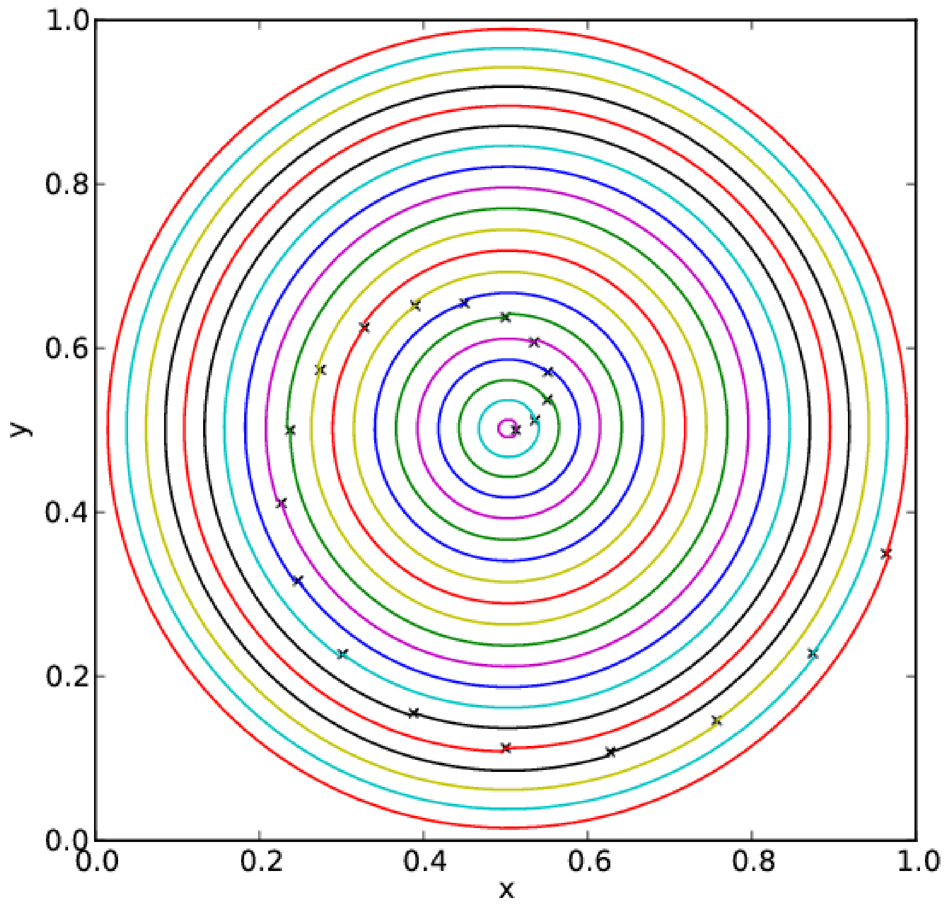
Unit Testing

- When writing a complex program (e.g. a simulation code), there can be many separate steps / solvers involved in getting your answer.
  - Finding out the source of errors in such a complicated code can be tough.
- Unit testing is the practice in which each smallest, self-contained unit of the code is tested independently of the others.
  - You could, for example, start by writing tests for each of the major physics units, and then worry about lower levels
- Implementation:
  - Either write your own simple driver for each routine to be tested
  - Unit testing frameworks automate some tasks
    - pytest is a very popular one—we'll see some examples of this at the end of the semester

Unit Testing

- Simple example: matrix inversion
  - Your code have a matrix inversion routine that computes A^{-1}
  - A unit test for this routine can be:
    - Pick a vector x
    - Compute $b = A x$
    - Compute $x = A^{-1} b$
    - Does the x you get match (to machine tol) the original x?
- More complicated example: a hydro program may consist of
  - Advection routines, EOS calls (and inverting the EOS), Particles, Diffusion, Reactions
  - Each of these can be tested alone
- There is a python unit testing framework called pytest—we can explore that in the discussion forum if there is interest.

Unit Testing



Test of particle advection in our low Mach hydro code, Maestro

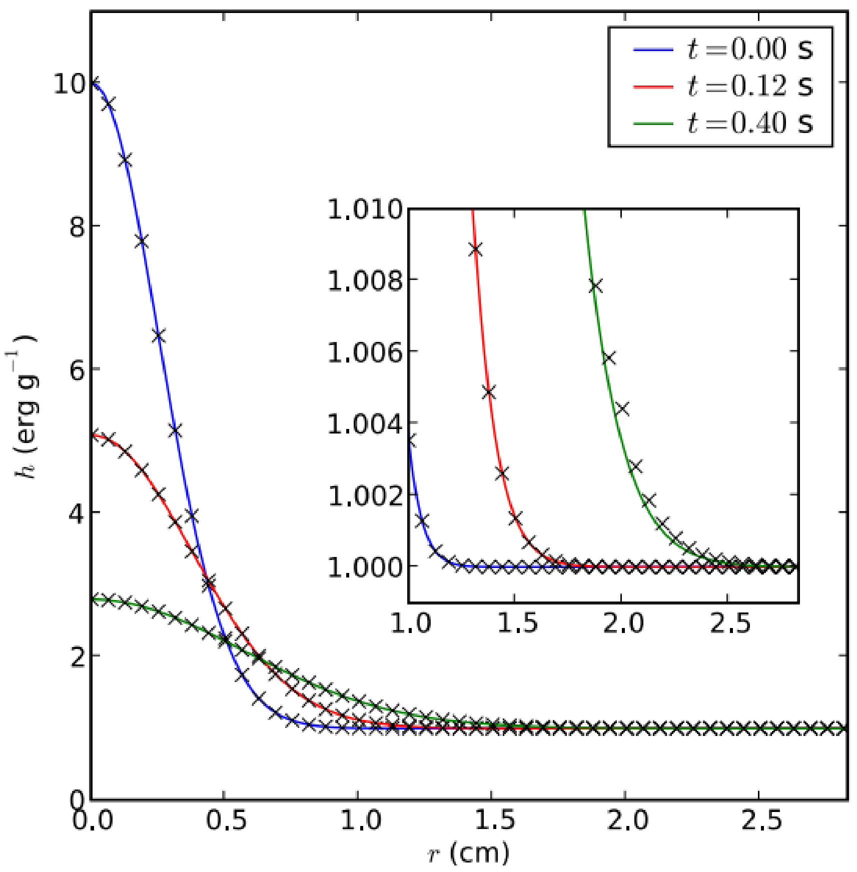


Figure 18. Average of enthalpy as a function of radius from the center, $(x, y) = (2.0, 2.0)$, of a two-dimensional Gaussian pulse. The $\times$'s are data from the numerical solution at the shown times. The lines represent the analytic solutions as given by Equation (A9). The numerical solution tracks the analytic solution very well except when the pulse has diffused enough that it begins to interact with the boundaries of the computational domain as seen in the inset plot.

Test of diffusion in our low Mach hydro code, Maestro

(Malone et al. 2011)

Unit Testing

- Whenever you important some legacy code into your project, write a unit test
  - Verifies that it performs as the authors intend
  - Allows for tests to interface changes you make to the code

General Rules

- When you write code, think to yourself: “if I come back to this 6 months from now, while I understand what I've done?”
  - If not, take the time now to make things clearer, document (even a simple README) what you've done, where the equations come from, etc.
  - You'll be surprised and how long your code lives on!
- Some languages let you do cute tricks.
  - Even if they might offer a small speed bump, if they complicate the code a lot to the point that it is hard to follow, then they're probably not worth it.
- Get things working before obsessing on performance

Automating Reproducibility

- Store meta-data in your output files that tell you where, when, what, and how the data was produced.
  - Already saw the example of the git hash in the makefile examples
  - Siesta example…

```
Siesta Version: siesta-trunk-462
Architecture  : x86_64-unknown-linux-gnu--unknown
Compiler flags: mpif90 -g -O2
PP flags      : -DMPI -DFC_HAVE_FLUSH -DFC_HAVE_ABORT
PARALLEL version

* Running on   24 nodes in parallel
>> Start of run:  15-AUG-2018   1:06:51

                           ***********************
                           *  WELCOME TO SIESTA  *
                           ***********************

reinit: Reading from standard input
************************** Dump of input data file
****************************
SystemName      Strontium titanate (SrTiO3)
SystemLabel     srtio3
```

```
FCOMP:            gfortran
FCOMP version:    gcc version 4.7.2 20121109 (Red Hat 4.7.2-8) (GCC)

F90 compile line: mpif90  -Jt/Linux.gfortran.debug.mpi/m -It/Linux.gfortran.debug.mpi/m -g -fno-range-
check -O1 -fbounds-check -fbacktrace -Wuninitialized -Wunused -ffpe-trap=invalid -finit-real=nan  -I..
/../../Microphysics/EOS/helmeos  -c

F77 compile line: gfortran   -Jt/Linux.gfortran.debug.mpi/m -It/Linux.gfortran.debug.mpi/m -g -fno-ran
ge-check -O1 -fbounds-check -fbacktrace -Wuninitialized -Wunused -ffpe-trap=invalid -finit-real=nan  -
I../../../Microphysics/EOS/helmeos  -c

C compile line:   gcc  -std=c99 -Wall -g -O1 -DBL_Linux -DBL_FORT_USE_UNDERSCORE   -c

linker line:      mpif90  -Jt/Linux.gfortran.debug.mpi/m -It/Linux.gfortran.debug.mpi/m -g -fno-range-
check -O1 -fbounds-check -fbacktrace -Wuninitialized -Wunused -ffpe-trap=invalid -finit-real=nan  -I..
/../../Microphysics/EOS/helmeos


===============================================================================
 Grid Information
===============================================================================
 level:           1
    number of boxes =          60
    maximum zones   =         384         640

 Boundary Conditions
   -x: periodic
   +x: periodic

   -y: slip wall
   +y: outlet


===============================================================================
 Species Information
===============================================================================
index                name           short name        A        Z
-------------------------------------------------------------------------------
    1     carbon-12                     C12      12.00     6.00
    2     oxygen-16                     O16      16.00     8.00
    3     magnesium-24                  Mg24     24.00    12.00
```

+ values of all runtime parameters…

Debuggers

- Simplest debugging: lots of prints!
- Interactive debuggers let you step through your code line-by-line, inspect the values of variables as they are set, etc.
- pdb is the python debugger
- If you just want to know how the code gets to a certain function:

```
import traceback
traceback.print_stack()
```

Commenting and Documentation

- The only thing worse than no comments are wrong comments
  - Comments can easily get out of date as code evolves
- Comments should convey to the reader the basic idea of what the next set of lines will accomplish.
  - Avoid commenting obvious steps if you've already described the basic idea
- Many packages allow for automatic documentation of routines/interfaces using pragmas put into the code as comments.

Source Code Libraries

- There are many sources for open, well-tested, published codes that may already do what you want.
  - This makes it easier to get going, may offer better algorithms than you were prepared to code.
  - Benefits from a community of developers and maturity
  - Still need to test, examine return codes, etc.
- Many of these mature codes are already wrapped for you in SciPy.
- For other codes, we'll look next at how to extend python in Fortran and C

Summary

- Some basic coding practices can greatly improve the reliability of your code
  - Frees you to do science
- Small learning curve is greatly offset by the improved productivity and stability