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.

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 mentioned: 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`

with

- `with` uses a context manager that has a special `__enter__` and `__exit__` function.
- Simplifies some common constructions
- Eg.

```
with open("x.txt") as f:
    data = f.read()
    # do something with data
```

  - This will open the file, return the file object `f`, allow you to work with it, and close the file automatically when this block is over

http://effbot.org/zone/python-with-statement.htm

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
  - Fun trick: LibreOffice files are zipped XML, but you can have it store the output as uncompressed “flat” XML files (.fodp instead of .odp)

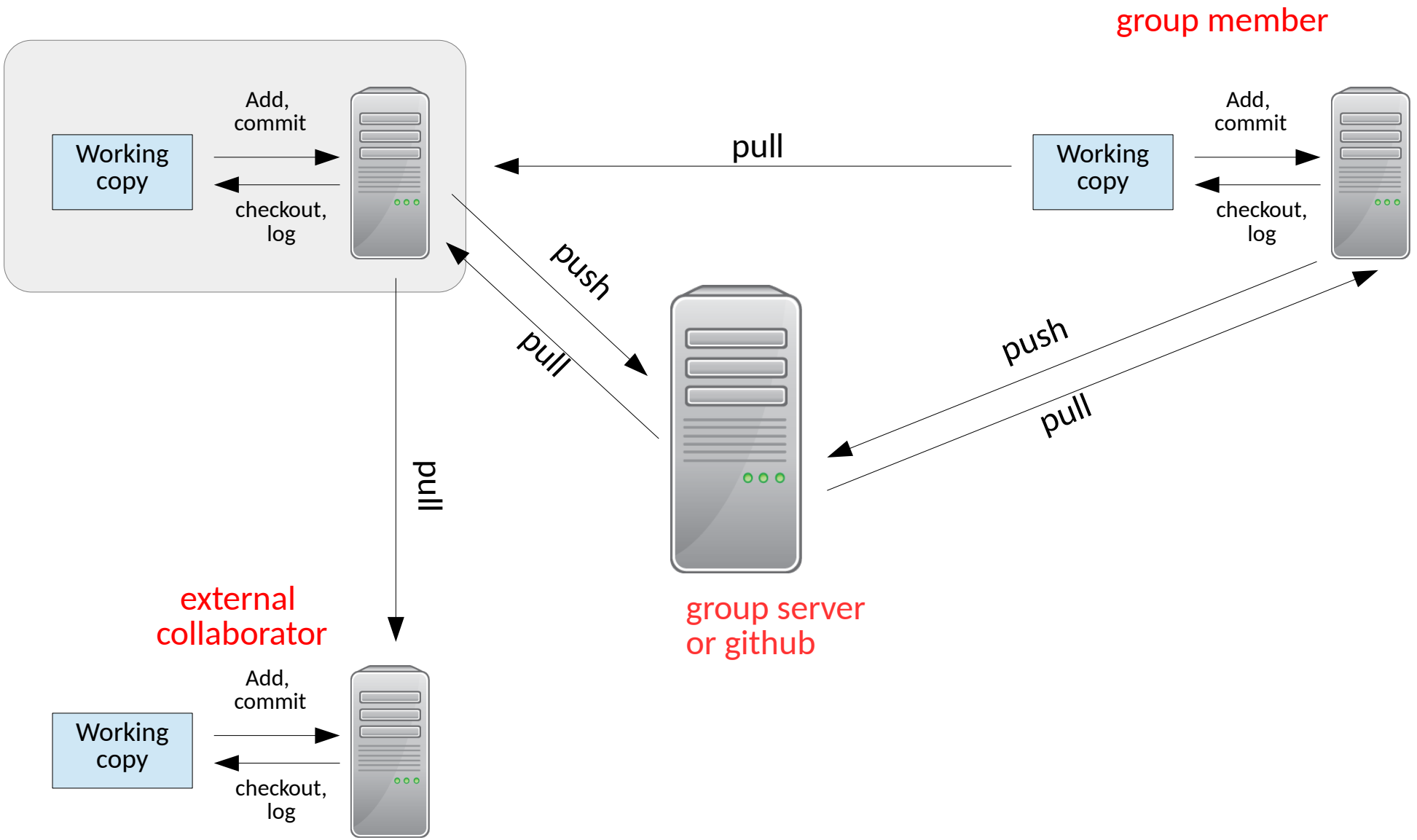
Centralized vs. Distributed Version Control

- 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)

Quick git Example

- We'll look at the example of having people work with a shared remote repository—this is common with groups.
  - Each developer will have their own clone that they interact with, develop in, branch for experimentation, etc.
  - You can push and pull to/from the remote repo to stay in sync with others
  - You probably want to put everyone in the same UNIX group on the server
- We'll start by creating a shared master bare repo:
  - `git init --bare --shared myproject.git`
  - `chgrp -R groupname myrepo.git`

Note the permissions set the sticky bit for the group (guid)

Quick git Example

- This repo is empty, and bare—it will only contain the git files, not the actual source files you want to work on
- Each user should clone it
  - In some other directory. User A does:
    - `git clone /path/to/myproject.git`
  - Now you can operate on it
    - Create a file (README)
    - Add it to your repo: `git add README`
    - Commit it to your repo: `git commit README`
    - Push it back to the bare repo: `git push`
  - Note that for each commit you will be prompted to add a log message detailing the change

* older versions of git won't know where push to. Instead of this, you can tell git to use the proposed new (git 2.0) behavior by doing:

```
git config --global push.default simple
git push
```

Quick git Example

- If you get confused about where the remote repo you are working with is, you can do:
  - `git remote -v`

Quick git Example

- Now user B comes along and wants to play too:
  - In some other directory. User B does:
    - `git clone /path/to/myrepo.git`
  - Note that they already have the README file
    - Edit README
    - Commit you changes locally: `git commit README`
    - Push it back to the bare repo: `git push`
- Now user A can get this changes by doing: `git pull`
  - Note that I did this on my laptop for demonstration, but the different users can be on completely different machines (and in different countries), as long as they have access to the same server
  - In general, you can push to a *bare repo*, but you can pull from anyone

Quick git Example

- You can easily look at the history by doing: `git log`
- You can checkout an old version using the hash:
  - `git checkout hash`
  - Make changes, use this older version
  - Look at the list of branches: `git branch`
  - Switch back to the tip: `git checkout master`
- Other useful commands:
  - `git diff`
  - `git status`
  - Branching
    - `git branch experiment`
    - `git checkout experiment`

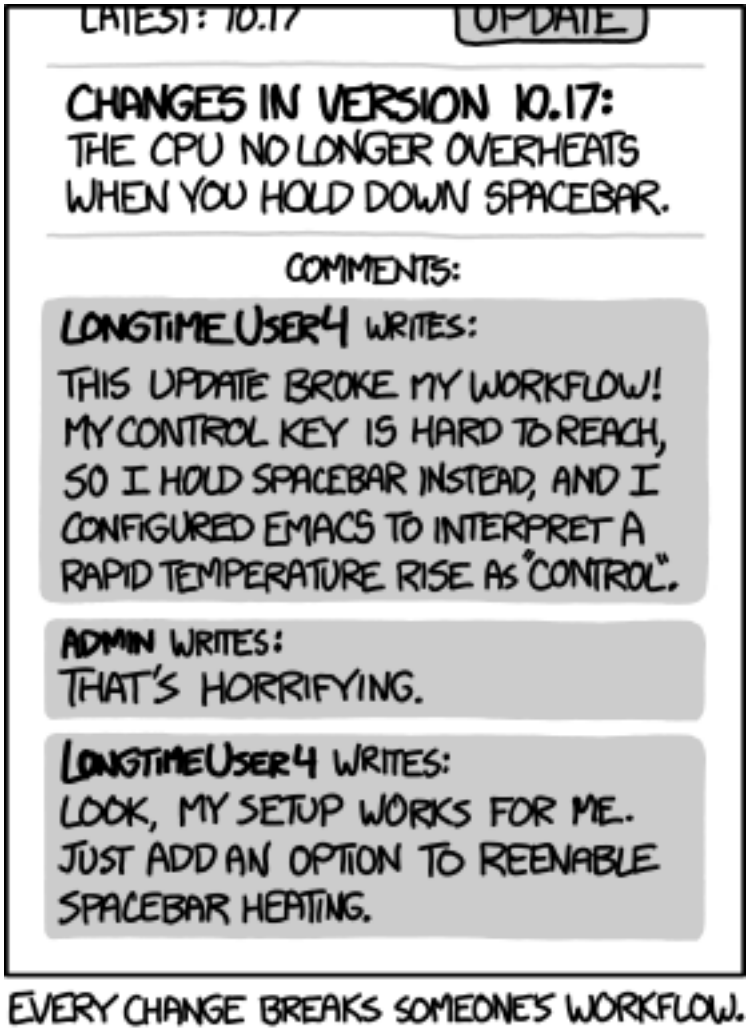
    } `git checkout -b experiment`
  - `git blame`

Quick git Example

- You can also put a link to your bare repo on the web and people can clone it remotely
  - Note you need to do `git update-server-info -f` after each change to the repo

Community

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



(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 request* and *issues*

Slack Integration

- You can integrate your research group's github into your slack
  - This is done in our git channel
  - Any changes, PRs, issues, etc. will be reported

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

What Kinds of Testing Exist?

- This is a big topic:
  - https://en.wikipedia.org/wiki/Software_testing
- We most commonly hear about:
  - Unit testing
    - Tests that a single function does what it was designed to do
  - Integration testing
    - Tests whether the individual pieces work together as intended
    - Sometimes done one piece at a time (iteratively)
  - Regression testing
    - Checks whether changes have changed answers
  - Verification & Validation (from the science perspective)
    - Verification: are we solving the equations correctly?
    - Validation: are we using the right equations in the first place?

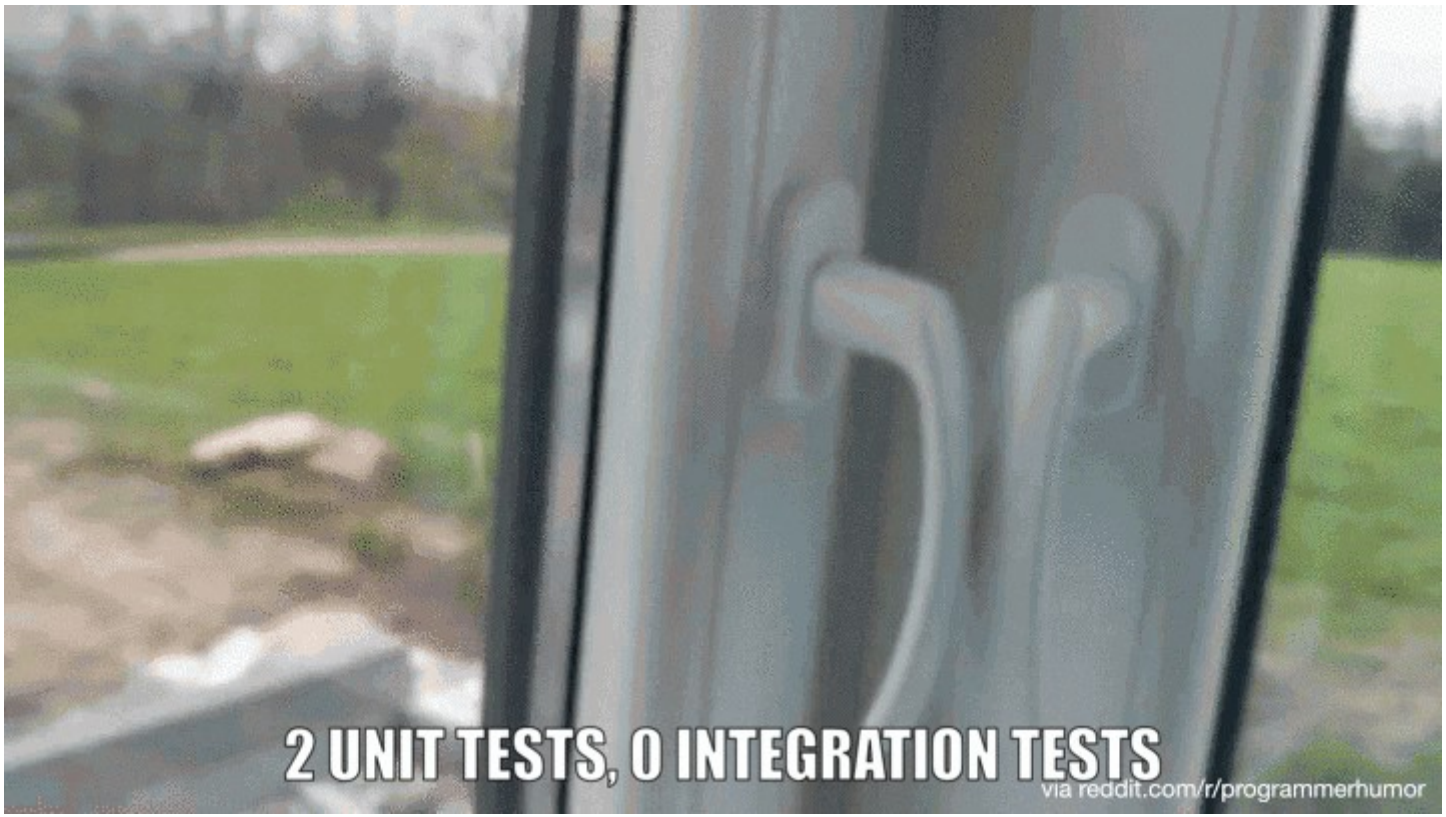
Automation

- The best testing is automated—you don't need to remember to run it
  - Developers can forget to manually run tests
- Github / bitbucket have *continuous integration* service that can be run on pull requests
  - This is commonly used on large development projects

Unit Testing

- We want to test the smallest piece of functionality alone
  - E.g., a simple test that ensures that a function does what it is designed to do
- Unit testing alone is not enough
  - Although it gives you confidence that each piece works alone as designed, it doesn't test what happens when you put it all together
  - *Integration testing* is a much harder problem

Unit vs. Integration Testing



Unit Testing

- When to write tests?
  - Some people advocate writing a unit test for a specification before you write the functions they will test
    - This is called Test-driven development (TDD): https://en.wikipedia.org/wiki/Test-driven_development
    - This helps you understand the interface, return values, side-effects, etc. of what you intend to write
  - Often we already have code, so we can start by writing tests to cover some core functionality
    - Add new tests when you encounter a bug, precisely to ensure that this bug doesn't arise again
- Tests should be short
  - You want to be able to run them frequently

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 nose—we can explore that in the discussion forum if there is interest.

Unit Testing

- There are several frameworks for unit testing in python—we'll focus on `pytest`
  - Note: like packaging, unit testing frameworks in python appear in a state of flux
  - For a long time, `nose` was one of the most popular frameworks, but its development has ceased
    - `nose2` is a successor to nose that may become standard soon
  - `unittest` is built into python, and may interact with nose2 in the future
  - `pytest` (sometimes `py.test`) seems to be the most popular now—we'll explore that
- Basic elements:
  - Discoverability: it will find the tests
  - Automation
  - Fixtures (setup and teardown)

pytest Unit Testing

- Install (single-user) via: `pip3 install -U pytest –user`
  - `pytest` should now be in your path
    - Note: older name was `py.test`
  - For user-install, it may be that `~/.local/bin/` comes later in your PATH than a system-wide directory with an older `pytest`
    - You may need to explicitly alias it
- For coverage reports: `pip3 install -U pytest-cov --user`

pytest Unit Testing

- Test discovery makes unit testing easy—adhere to these conventions and your tests will be found:
  - File names should start or end with "`test`":
    - `test_example.py`
    - `example_test.py`
  - For tests in a class, the class name should begin with "`Test`"
    - e.g., `TestExample`
    - There should be no `__init__()`
  - Test method / function names should start with "`test_`"
    - e.g., `test_example()`

https://pytest.readthedocs.io/en/reorganize-docs/new-docs/user/naming_conventions.html

pytest Unit Testing

- Tests use assertions (via python's `assert` statement) to check behavior at runtime
  - https://docs.python.org/3/reference/simple_stmts.html#assert
  - Basic usage: `assert expression`
    - Raises `AssertionError` if `expression` is not true
    - e.g., `assert 1 == 0` will fail with an exception

Simple pytest Example

- Here's a simple example (`day-4/simple/` in our git repo):

```
def multiply(a, b):
    return a*b

def test_multiply():
    assert multiply(4, 6) == 24

def test_multiply2():
    assert multiply(5, 6) == 24
```

  - There are 2 tests here
    - First will pass, second will fail
  - Run the tests as: `pytest -v .`

pytest Unit Testing

- Class example
  - Note, a class used for testing is not a full-fledged class—it simply helps to organize data used for a bunch of tests with common needs
  - In particular, it does not have a constructor (`__init__()`)
    - See, e.g., https://stackoverflow.com/questions/21430900/py-test-skips-test-class-if-constructor-is-defined
- We'll look at an example with a NumPy array
  - `examples/testing/pytest/class/` in our class git
  - We always want the array to exist for our tests, so we'll use fixtures (in particular `setup_method()`) to create the array
  - Using a class means that we can access the array created in setup from our class
- NumPy has its own assertion functions
  - https://docs.scipy.org/doc/numpy/reference/routines.testing.html
  - Note in particular, the approximately equal tests

pytest Coverage

- We can determine the coverage of our testing
  - Note just because a line / function is covered in our suite, doesn't mean we'll capture every error—there can always be corner-cases, things you don't anticipate
  - Adding unit tests is an ongoing process
- Basic running:
  - `pytest --cov .`
  - Here, '`.`' is the path we are testing
- Detailed report (lines missed):
  - `pytest --cov-report term-missing --cov .`

pytest Coverage

- Let's look at a larger example:
  - pyro is my tutorial hydrodynamics code: https://github.com/zingale/pyro2
- pyro has both unit tests (via pytest) and regression testing (more on that later)
- Tests are stored in the various `tests/` subdirectories

```
.
├── advection
│   ├── problems
│   └── tests
├── advection_rk
│   ├── problems -> ../advection/problems
│   └── tests
├── analysis
├── compressible
│   ├── problems
│   └── tests
├── compressible_react
├── compressible_rk
│   ├── problems -> ../compressible/problems
│   └── tests
├── diffusion
│   ├── problems
│   └── tests
├── incompressible
│   ├── problems
│   └── tests
├── lm_atm
│   ├── problems
│   └── tests
├── lm_combustion
├── mesh
│   ├── experiments
│   └── tests
├── multigrid
│   └── tests
├── radhydro
├── util
│   └── tests
└── www
```

Regression Testing

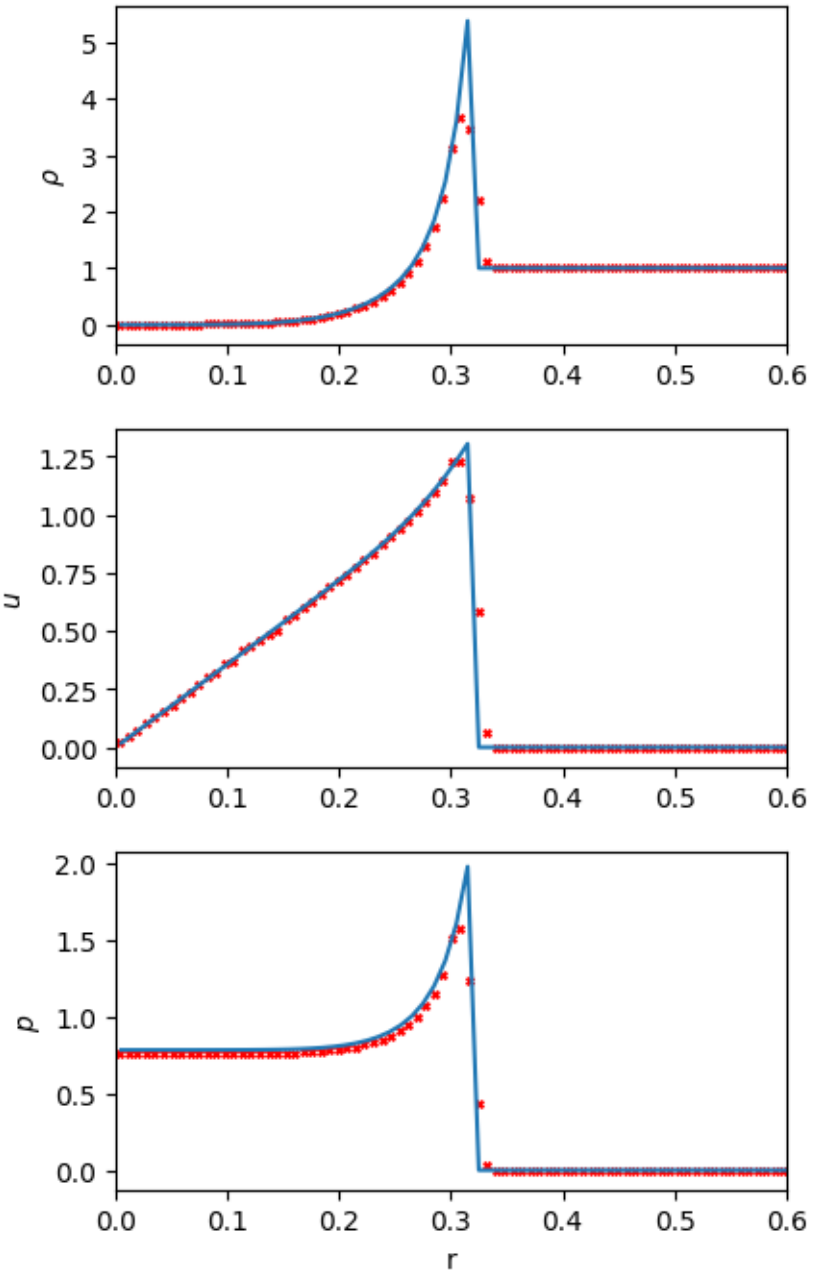
- Basic idea:
  - Pick some problems / workflows that exercise your codebase
    - Might need many tests to get good coverage / explore all options
  - Store a benchmark containing the “right” answer
  - Each time you change your code, run the regression tests
  - Compare the new answer to the stored benchmark
- If a regression test fails, then either:
  - You’ve introduced a bug—look at what changed and fix it
  - You fixed a bug—update the benchmarks

Regression Testing

- Automating the testing
  - You need a tool to do the comparisons
  - Store benchmarks in a separate directory (so they are not overwritten when you run)
  - Run your code, at the end of the run, compare the new output to the stored benchmark and report
- Our example, pyro, has regression testing built in with the `--compare_benchmark` option
- Here's another example from my research codes: http://bender.astro.sunysb.edu/Castro/test-suite/test-suite-gfortran/

Verification

- For science codes, *verification* means test our algorithm against a known solution to the system of equations we are solving
  - This doesn't help us understand if we picked the right equations (e.g., correct model of reality) to begin with
- For the equations of hydrodynamics, the Sedov blast wave problem is a common verification problem
  - A lot of energy is put into a small volume in a uniform domain
  - Spherical blast wave develops
  - Analytic solution found in the 1940s



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
  - Maestro example…

```
================================================================================
 Job Information
================================================================================
job name:
inputs file: inputs_2d

number of MPI processes      1
number of threads            1

CPU time used since start of simulation (CPU-hours)     0.1473997255E-01


================================================================================
 Plotfile Information
================================================================================
output date:              2013-05-08
output time:              11:39:43
output dir:               /home/zingale/development/MAESTRO/Exec/TEST_PROBLEMS/test2
time to write plotfile (s)    0.5483140945


================================================================================
 Build Information
================================================================================
build date:    2013-05-08 11:38:04.553714
build machine: Linux nan.astro.sunysb.edu 3.8.9-200.fc18.x86_64 #1 SMP Fri Apr 26 12:50:07 UTC 2013 x8
6_64 x86_64 x86_64 GNU/Linux
build dir:     /home/zingale/development/MAESTRO/Exec/TEST_PROBLEMS/test2
BoxLib dir:    /home/zingale/development/BoxLib/

MAESTRO    git hash: bad8ea8d66871a2172dcb276643edce53f739695
BoxLib     git hash: febf34dba7cc701a78c73e16a543f062cf36d587
AstroDev   git hash: 5316edb829577f80977fd2db8a113ccc4da42e02

modules used:
  Util/model_parser
  Util/random
  Util/VODE
  Util/BLAS
  Source
  ../../../Microphysics/EOS
  ../../../Microphysics/EOS/helmeos
  ../../../Microphysics/networks/ignition_simple
```

```
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…

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.