Missing Semester Lecture 6 Notes

Build systems are tools that we use to do a lot of repeated build work
- **Metaprogramming** = processes that surround working w/ software
- We often have **targets** (things we want to build) w/ **dependencies**
- Think about all the commands that you run to "build" something
- We need **rules** that determine how to build something in a sequence
- Here, we'll focus on **make**, a very common build tool for coding rules
- Make looks for a Makefile (in your current directory) with instructions
- Dependency: something that, when changed, necessitates a re-build
- Note that the rules in a Makefile tell EXACTLY how to get from A to B
- Make tries to *find* and/or *build* the dependencies you give it
- Remember: Make tries to do the minimal amount of work to build
- Note: if your Makefile gets huge, usually there's a better tool to use

Dependency management:
- In general, wherever there are packages, there are managers
- Note that versions are sometimes confusing (**Why do we version?**)
- Common **semantic versioning** structure: 8.1.7 (major.minor.patch)
- Increment patch when a change is TOTALLY backwards compatible
- Make a "minor release" when you ADD something, then set patch=0
- Make a "major release" after a backwards INcompatible change
- **Lockfiles**: lists of your dependencies and their current local versions
- These lockfiles can make builds quicker, and reproducible (useful)

Continuous Integration: a customized chain of event-triggered actions
- Some CI platforms are super open-ended, some come pre-filled-out
- You could even use CI to auto-update stuff in your project repo
- Badges are a thing w/ CI, give you a quick overview of a repo's status
- Cool: The MS website is built w/ GitHub pages + CI! (all markdown)
- Note: GitLab's CI pipeline system is pretty nice to use (OTSDocker)
- **CD** (continuous deployment) is a thing too! (Do we know what it is?)
- **Remember**: Computers are way better at repetitive stuff than we are!

Test Suites: Large collection of tests usually run together as a unit
- **Unit test**: small, self-contained, single-feature test
- **Integration test**: test interactions b/w different program subsystems
- **Regression test**: tests things that *were* broken (that they DNE now)
- **Mocking**: filling in dummy-versions at some steps to *isolate* a test
- Note: testing is NOT something that you want to spend your time on
- **Always** let the computer do the busy work so you can think bigger!!!!!

FOR LATER REVIEW AND EXERCISES:

Exercise 1:
Do we all understand what happened with the lecture example of make??
(i.e. please explain it to me)

Go somewhere where you don't mind to play around, and run:
git clone https://github.com/ldishman/make_demo.git

- Take a look ('ls') and review what everything is/does
- Understand what the Makefile does, and THEN run 'make'
- NOTE: If you have any issues with plot.py, edit the shebang line at the top to whatever python version you use

Now, go ahead and try to follow the instructions from exercise 1!

Exercise 1 Solution: (add this to your Makefile, then run 'make clean')

```
.PHONY: clean
clean:
  rm -f paper.pdf plot-*.png *.aux *.log
```

Note: For our session, not going to worry about 'git ls-files', but feel free to implement this idea on your own time!

Exercise 2: This is pretty straightforward, and we'll use it as a way to make sure we all understand the major.minor.patch versioning syntax

Custom Exercise: Let's take a look at the OTSDocker repo on GitLab!
Promo: any of you guys could get involved with this if you're interested
Quickly talk through the usefulness of git with this project (last lecture)
Demonstrate what a **CI pipeline** looks like, and what triggers one

Exercises 3-5: You should try these on your own time!!! They will be super useful, we just won't have time to do them together :(