The Unix Shell

(index.html)

Instructor's Guide

- Why do we learn to use the shell?
  - Allows users to automate repetitive tasks
  - And capture small data manipulation steps that are normally not recorded to make research reproducible
- The Problem
  - Running the same workflow on several samples can be unnecessarily labour intensive
  - Manual manipulation of data files:
    - is often not captured in documentation
    - is hard to reproduce
    - is hard to troubleshoot, review, or improve
- The Shell
  - Workflows can be automated through the use of shell scripts
  - Built-in commands allow for easy data manipulation (e.g. sort, grep, etc.)
  - Every step can be captured in the shell script and allow reproducibility and easy troubleshooting

Overall

Many people have questioned whether we should still teach the shell. After all, anyone who wants to rename several thousand data files can easily do so interactively in the Python interpreter, and anyone who's doing serious data analysis is probably going to do most of their work inside the IPython Notebook or R Studio. So why teach the shell?

The first answer is, "Because so much else depends on it." Installing software, configuring your default editor, and controlling remote machines frequently assume a basic familiarity with the shell, and with related ideas like standard input and output. Many tools also use its terminology (for example, the `%ls` and `%cd` magic commands in IPython).

The second answer is, "Because it's an easy way to introduce some fundamental ideas about how to use computers." As we teach people how to use the Unix shell, we teach them that they should get the computer to repeat things (via tab completion, `!` followed by a command

number, and `for` loops) rather than repeating things themselves. We also teach them to take things they've discovered they do frequently and save them for later re-use (via shell scripts), to give things sensible names, and to write a little bit of documentation (like comment at the top of shell scripts) to make their future selves' lives better.

The third answer is, "Because it enables use of many domain-specific tools and compute resources researchers cannot access otherwise." Familiarity with the shell is very useful for remote accessing machines, using high-performance computing infrastructure, and running new specialist tools in many disciplines. We do not teach HPC or domain-specific skills here but lay the groundwork for further development of these skills. In particular, understanding the syntax of commands, flags, and help systems is useful for domain specific tools and understanding the file system (and how to navigate it) is useful for remote access.

Finally, and perhaps most importantly, teaching people the shell lets us teach them to think about programming in terms of function composition. In the case of the shell, this takes the form of pipelines rather than nested function calls, but the core idea of "small pieces, loosely joined" is the same.

All of this material can be covered in three hours as long as learners using Windows do not run into roadblocks such as:

- not being able to figure out where their home directory is (particularly if they're using Cygwin);
- not being able to run a plain text editor; and
- the shell refusing to run scripts that include DOS line endings.

Teaching Notes

- Time estimates:
    - @gvwilson: 3 hours
- Super cool online resource! http://explainshell.com/ will dissect any shell command you type in and display help text for each piece.
- Resources for "splitting" your shell so that recent commands remain in view: https://github.com/rgaiacs/swc-shell-split-window
- How to use the materials in the shell-novice repository (or, lesson planning)
    - For a great general list of tips, see this swcarpentry blog post (http://software-carpentry.org/blog/2015/03/teaching-tips.html)
    - Use the `data` directory for in-workshop exercises and live coding examples. You can clone the shell-novice directory or use the `Download ZIP` button on the right to get the entire repository; we also now provide a zip file of the `data` directory that can be downloaded on its own from the repository by right-click + save. See the "Get Ready" box on the front page of the website for more details.
    - Website: various practices have been used.
        - Option 1: Can give links to learners before the lesson so they can follow along, catch up, and see exercises (particularly if you're following the lesson content without many changes).
        - Option 2: Don't show the website to the learners during the lesson, as it can be distracting - students may read instead of listen, and having another

window open is an additional cognitive load.
        - In any case, make sure to point to website as a post-workshop reference.
    - Content: Unless you have a truly generous amount of time (4+ hours), it is likely that you will not cover ALL the material in this lesson in a single half-day session. Plan ahead on what you might skip, what you really want to emphasize, etc.
    - Exercises: Think in advance about how you might want to handle exercises during the lesson. How are you assigning them (website, slide, handout)? Do you want everyone to try it and then you show the solution? Have a learner show the solution? Have groups each do a different exercise and present their solutions?
    - `reference.md` can be printed out and given to students as a reference, your choice.
    - Other preparation: Feel free to add your own examples or side comments, but know that it shouldn't be necessary - the topics and commands can be taught as given on the lesson pages! If you think there is a place where the lesson is lacking, feel free to raise an issue (https://github.com/swcarpentry/shell-novice/issues) or submit a pull request (https://github.com/swcarpentry/shell-novice/pulls).
- Setup:
    - Run `tools/gen-nene.py` to regenerate random data files if needed (some are already in the `filesystem` directory).
    - Run `tools/gen-sequence.py` to regenerate random sequence data if needed.
- Have learners open a shell and then do `whoami`, `pwd`, and `ls`. Then have them create a directory called `workshop` and `cd` into it, so that everything else they do during the lesson is unlikely to harm whatever files they already have.

- Get them to run an editor and save a file in their `workshop` directory as early as possible. Doing this is usually the biggest stumbling block during the entire lesson: many will try to run the same editor as the instructor (which may leave them trapped in the awful nether hell that is Vim), or will not know how to navigate to the right directory to save their file, or will run a word processor rather than a plain text editor. The quickest way past these problems is to have more knowledgeable learners help those who need it.

- Tab completion sounds like a small thing: it isn't. Re-running old commands using `!123` or `!wc` isn't a small thing either, and neither are wildcard expansion and `for` loops. Each one is an opportunity to repeat one of the big ideas of Software Carpentry: if the computer can repeat it, some programmer somewhere will almost certainly have built some way for the computer to repeat it.

- Building up a pipeline with four or five stages, then putting it in a shell script for re-use and calling that script inside a `for` loop, is a great opportunity to show how "seven plus or minus two" connects to programming. Once we have figured out how to do something moderately complicated, we make it re-usable and give it a name so that it only takes up one slot in working memory rather than several. It is also a good opportunity to talk about exploratory programming: rather than designing a program up front, we can do a few useful things and then retroactively decide which are worth encapsulating for future re-use.

- If everything is going well, you can drive home the point that file extensions are essentially there to help computers (and human readers) understand file content and are not a requirement of files (covered briefly in Files and Directories (01-filedir.html)). This can be done in the Pipes and Filters (03-pipefilter.html) section by showing that you can

redirect standard output to a file without the .txt extension (e.g., lengths), and that the resulting file is still a perfectly usable text file. Make the point that if double-clicked in the GUI, the computer will probably ask you what you want to do.

- We have to leave out many important things because of time constraints, including file permissions, job control, and SSH. If learners already understand the basic material, this can be covered instead using the online lessons as guidelines. These limitations also have follow-on consequences:
- It's hard to discuss `#!` (shebang) without first discussing permissions, which we don't do.
- Installing Bash and a reasonable set of Unix commands on Windows always involves some fiddling and frustration. Please see the latest set of installation guidelines for advice, and try it out yourself before teaching a class.
- On Windows, it appears that:

  ```
  $ cd
  $ cd Desktop
  ```

  will always put someone on their desktop. Have them create the example directory for the shell exercises there so that they can find it easily and watch it evolve.
- Stay within POSIX-compliant commands, as all the teaching materials do. Your particular shell may have extensions beyond POSIX that are not available on other machines, especially the default OSX bash and Windows bash emulators. For example, POSIX `ls` does not have an `--ignore=` or `-I` option, and POSIX `head` takes `-n 10` or `-10`, but not the long form of `--lines=10`.

Windows

Installing Bash and a reasonable set of Unix commands on Windows always involves some fiddling and frustration. Please see the latest set of installation guidelines for advice, and try it out yourself before teaching a class. Options we have explored include:

1. msysGit (http://msysgit.github.io/) (also called "Git Bash"),
2. Cygwin (http://www.cygwin.com/),
3. using a desktop virtual machine, and
4. having learners connect to a remote Unix machine (typically a VM in the cloud).

Cygwin was the preferred option until mid-2013, but once we started teaching Git, msysGit proved to work better. Desktop virtual machines and cloud-based VMs work well for technically sophisticated learners, and can reduce installation and configuration at the start of the workshop, but:

1. they don't work well on underpowered machines,
2. they're confusing for novices (because simple things like copy and paste work differently),
3. learners leave the workshop without a working environment on their operating system of choice, and
4. learners may show up without having downloaded the VM or the wireless will go down (or become congested) during the lesson.

Whatever you use, please test it yourself on a Windows machine before your workshop: things may always have changed behind your back since your last workshop. And please also make use of our Windows setup helper.
