The Unix Shell

(index.html)

Introducing the Shell

Learning Objectives

- Explain how the shell relates to the keyboard, the screen, the operating system, and users' programs.
- Explain when and why command-line interfaces should be used instead of graphical interfaces.

At a high level, computers do four things:

- run programs
- store data
- communicate with each other
- interact with us

They can do the last of these in many different ways, including direct brain-computer links and speech interfaces. Since these are still in their infancy, most of us use windows, icons, mice, and pointers. These technologies didn't become widespread until the 1980s, but their roots go back to Doug Engelbart's work in the 1960s, which you can see in what has been called "The Mother of All Demos (http://www.youtube.com/watch?v=a11JDLBXtPQ)".

Going back even further, the only way to interact with early computers was to rewire them. But in between, from the 1950s to the 1980s, most people used line printers. These devices only allowed input and output of the letters, numbers, and punctuation found on a standard keyboard, so programming languages and interfaces had to be designed around that constraint.

This kind of interface is called a `command-line interface`, or CLI, to distinguish it from a `graphical user interface`, or GUI, which most people now use. The heart of a CLI is a `read-evaluate-print loop`, or REPL: when the user types a command and then presses the enter (or return) key, the computer reads it, executes it, and prints its output. The user then types another command, and so on until the user logs off.

This description makes it sound as though the user sends commands directly to the computer, and the computer sends output directly to the user. In fact, there is usually a program in between called a `command shell`. What the user types goes into the shell, which then figures out what commands to run and orders the computer to execute them. Note, the shell is called the shell because it encloses the operating system in order to hide some of its complexity and make it simpler to interact with.

A shell is a program like any other. What's special about it is that its job is to run other programs rather than to do calculations itself. The most popular Unix shell is Bash, the Bourne Again SHell (so-called because it's derived from a shell written by Stephen Bourne — this is what passes for wit among programmers). Bash is the default shell on most modern implementations of Unix and in most packages that provide Unix-like tools for Windows.

Using Bash or any other shell sometimes feels more like programming than like using a mouse. Commands are terse (often only a couple of characters long), their names are frequently cryptic, and their output is lines of text rather than something visual like a graph. On the other hand, the shell allows us to combine existing tools in powerful ways with only a few keystrokes and to set up pipelines to handle large volumes of data automatically. In addition, the command line is often the easiest way to interact with remote machines and supercomputers. Familiarity with the shell is near essential to run a variety of specialised tools and resources including high-performance computing systems. As clusters and cloud computing systems become more popular for scientific data crunching, being able to interact with them is becoming a necessary skill. We can build on the command-line skills covered here to tackle a wide range of scientific questions and computational challenges.

Nelle's Pipeline: Starting Point

Nelle Nemo, a marine biologist, has just returned from a six-month survey of the North Pacific Gyre (http://en.wikipedia.org/wiki/North_Pacific_Gyre), where she has been sampling gelatinous marine life in the Great Pacific Garbage Patch (http://en.wikipedia.org/wiki/Great_Pacific_Garbage_Patch). She has 300 samples in all, and now needs to:

1. Run each sample through an assay machine that will measure the relative abundance of 300 different proteins. The machine's output for a single sample is a file with one line for each protein.
2. Calculate statistics for each of the proteins separately using a program her supervisor wrote called `goostat`.
3. Compare the statistics for each protein with corresponding statistics for each other protein using a program one of the other graduate students wrote called `goodiff`.
4. Write up results. Her supervisor would really like her to do this by the end of the month so that her paper can appear in an upcoming special issue of Aquatic Goo Letters.

It takes about half an hour for the assay machine to process each sample. The good news is that it only takes two minutes to set each one up. Since her lab has eight assay machines that she can use in parallel, this step will "only" take about two weeks.

The bad news is that if she has to run `goostat` and `goodiff` by hand, she'll have to enter filenames and click "OK" 45,150 times (300 runs of `goostat`, plus 300x299/2 runs of `goodiff`). At 30 seconds each, that will take more than two weeks. Not only would she miss her paper

deadline, the chances of her typing all of those commands right are practically zero.

The next few lessons will explore what she should do instead. More specifically, they explain how she can use a command shell to automate the repetitive steps in her processing pipeline so that her computer can work 24 hours a day while she writes her paper. As a bonus, once she has put a processing pipeline together, she will be able to use it again whenever she collects more data.
