Tutorial 1 Session 2 - Using Unix commands

Before we start
Session 1: Everything is somewhere - hosts, networks and filesystems
Session 2: The computer as a nut - using Unix commands
The computer as a nut

In session 1 we mentioned briefly mentioned the shell. This is part of a bigger analogy: A Unix system is like a nut.

You occasionally hear Unix-oriented people talking about the kernel - this is the heart of the operating system, how the software interfaces with the hardware. If you write software, when you use a print statement, or open and read or write a file, the program makes a call to the kernel which
manages the interactions with the disk and the screen. As users, we mostly don't need to care about the kernel.

Whenever we type commands into a terminal, we are working in the shell. The job of the shell is to interpret user input, run programs accordingly and report back to the user. There are a number of different shells available, of which bash is the default on Unix system. The GUIs provided by OXS and Windows are also shells, albeit much less flexible ones than command-line shells such as bash.

The part between the shell and the kernel - the meat, if you like - are the commands and programs that do the actual computational work. Everything from “cd” to Matlab or a complex CFD model sits here.

In this session, we’ll learn how to use the shell and a few of the essential commands that make up a Unix system.

**Don’t Panic**
There’s a lot to learn about the shell, but you don’t need to learn it all at once. We’ll cover a lot here, but you don’t need to remember it all - the aim of this session is to provide an overview of a few key concepts, and to try using them yourself. You can always come back for another look, or contact us for help, and there is plenty more information on the Internet. So relax, open a terminal and have a go!

**It's 2016. Why use the command line? Isn't there a GUI?**

Granted, the command line has been around for a lot longer than graphical interfaces. And graphical interfaces are simple and intuitive to use. But the expressiveness of point-and-click is limited to “click”, that is, “perform an action that somebody has prepared beforehand” - normally “open this file” or “launch this program”.

**Exercise**
Think about how you might achieve the following from a Windows or OSX GUI:

1. I'm running out of disk space/disk quota - what's using it all up? Show me which of my directories are taking the most space
2. Delete all of the compiled object files (but none of the source code) from this directory tree
3. I've realized that the set of batch jobs I submitted which are named “scenario_2a”, "scenario_2b", etc were set up incorrectly. I want to delete those jobs, but leave my other jobs running

None of these is terribly complicated, but they're all a bit too specialized for there to be an app for that, so each one will require a lot of pointing and clicking. It will get laborious very quickly.

Here's how you could do each on the Unix command line (don't worry too much about the specifics of the commands and piping, well get to that soon):

1. `du --max_depth=3 | sort -nr | head`
   - `du` shows the disk usage of a directory and all of it's subdirectories, we'll do that and then sort the results in decreasing total size, and print the top ten.
   (Note for Mac users: on OSX, instead of du use `du -d`)
2. `find . -name ".o" -delete`
   - A single command for finding files matching some criteria and doing something with them. You just need to specify the criteria and the action.
3. `qstat -u $USER | awk '/^[0-9]/ { if ( $4 ~ "scenario_2[a-z]\"" ) print $1 ; }' | cut -d. -f1 | xargs qdel`
   - List the jobs belonging to me, filter the list for those with a name from "scenario_2a" to "scenario_2z", get the job ids and pass those to `qdel` to be deleted.

The point here is that the **expressiveness of the command line gives you almost as much power and flexibility as a programming language**. The “meat” of the nut - the programs and commands - form a vocabulary and the shell provides a kind of grammar. **We're moving from pointing at things and grunting to articulating exactly what we want.** There's a learning curve, but it's worth it.

**The trouble with interactive environments**

There is another reason why GUIs are less common in HPC environments: **point-and-click is necessarily interactive.** In HPC environments (as we'll see in session 3) work is scheduled in order to allow exclusive use of the shared resources. On a busy system there may be several hours wait between when you submit a job and when the resources become available, so a reliance on user interaction is not viable. In Unix, commands need not be run interactively at the prompt, you can write a sequence of commands into a file to be run as a script, either manually (for sequences you find yourself repeating frequently) or by another program such as the batch system.

**The Unix Philosophy**
Unix has been successful for a long time, and its success is attributed mostly to its overall approach to computing. The examples above illustrate a few of the key tenets of this philosophy:

- Each program should do one thing, and do that one thing well
  Above, `du` simply reports the space used by directories in the order it finds them. Sorting the results is left to the `sort` command.

- Small is beautiful
  Notice that each command has a short, lowercase name (easy to type). Common switches are terse: `sort -nr` means "sort in numerical, reverse order".

- Everything is a filter
  Each program reads a stream of text, does something with it, and writes out a modified version of it. For example, `sort` writes the lines out in a specified order, `cut` removes sections of each line. So you can do complex things by inserting the relevant filter into the stream.
  Some useful names: the stream of text coming into a filter is called the standard input, or `stdin`. The stream of text leaving a filter is standard output, or `stdout` (in any programming language, `print` sends text to `stdout`). A third stream named `stderr` is also available for error messages.

- Use shell scripts to tie things together
  Each of these examples is like a small, ephemeral app - you have a bunch of tools and a way to link them together. By writing sequences of commands into a script you can perform even more complex workflows. This lego-like nature allows you to do things the developers of Unix didn't think of - the whole is more than the sum of its parts. (And if you're making tools that others will use, following this philosophy will allow your users to use your tools in ways you didn't think of either).

Next: Learning to speak - using the language of the shell