HPC Carpentry part 1 - Using the shell (./)

General Information

This workshop is an introduction to using high-performance computing systems effectively. We obviously can't cover every case or give an exhaustive course on parallel programming in just two days of teaching time. Instead, this workshop is intended to give students a good introduction and overview of the tools available and how to use them effectively.

By the end of this workshop, students will know how to:

- Connect to remote HPC systems and transfer data
- Use a scheduler to work on a shared system
- Use software modules to access different HPC software
- Work effectively on a remote shared resource

Instructors and helpers

- Mozhgan Kabiri chimeh (NVIDIA)
- Anna (Ania) Brown (University of Oxford, University of Southampton)
- Fouzhan Hosseini (Numerical Algorithms Group)
- Weronika Fillinger (EPCC)
- Neelofer Banglawala (EPCC)

Part 2

The material for the second part of this workshop is available at: https://aniabrown.github.io/hpc-carpentry-WHPC/ (https://aniabrown.github.io/hpc-carpentry-WHPC/).

Details

Who: The course is aimed at graduate students and other researchers. **You don't need to have any previous knowledge of the tools that will be presented at the workshop.**

Where: COM-G12-Main Lewin, Computer Science Department, Regent Court, 211 Portobello St, Sheffield. Get directions with OpenStreetMap (//www.openstreetmap.org/?mlat=53.381122&mlon=-1.479930&zoom=16) or Google Maps (//maps.google.com/maps?q=53.381122, -1.479930).

When: 09:00-17:00, 30 Jan 2020, 09:00-16:00 31 Jan 2020. Add to your Google Calendar. (//calendar.google.com/calendar/render?action=TEMPLATE&text=Software Carpentry Workshop&dates=20200130/20200131&trp=false&sprop&sprop=name:&sf=true&output=xml&location=COM-G12-Main Lewin, Computer Science Department, Regent Court, 211 Portobello St, Sheffield&details=Software Carpentry Workshop at COM-G12-Main Lewin)

Requirements: Participants must bring a laptop with a Mac, Linux, or Windows operating system (not a tablet, Chromebook, etc.) that they have administrative privileges on. They should have a few specific software packagesSetup (setup/) installed. They are also required to abide by Software Carpentry's Code of Conduct (https://software-carpentry.org/conduct.html)

Accessibility: We are committed to making this workshop accessible to everybody. The workshop organizers have checked that:

- The room is wheelchair / scooter accessible.
- Accessible restrooms are available.

Contact: Please email mozhgank@nvidia.com (mailto:mozhgank@nvidia.com) for more information.

Getting Started

To get started, follow the directions on the Setup (setup/) page to ensure you have the bash shell and an SSH client installed.

Pre workshop survey

To help us support you as best as possible during the workshop, we request that you please complete the following pre-course survey (https://forms.office.com/Pages/ResponsePage.aspx?id=JhaX55a5Lkazybu6OlDVXYnSk6_JQaBEovlJG8Ng14dUMEI2T0dNT1FSQ0FDV1lFWjVNTTE4Q1BEMC4u). It should only take a few moments.

We will use this collaborative document (https://pad.carpentries.org/2020-01-30-whpc-hpccarpentry) for chatting, taking notes, and sharing URLs and bits of code.

Schedule

	Setup (./setup/)	Download files required for the lesson
09:30	1. Why Use a Cluster? (./00-hpc-intro/index.html)	Why would I be interested in High Performance Computing (HPC)? What can I expect to learn from this course?
09:50	2. Connecting to the remote HPC system (./01-connecting/index.html)	How do I open a terminal? How do I connect to a remote computer?
10:20	3. Coffee Break (./011-coffee/index.html)	Break
10:50	4. Moving around and looking at things (./02-navigation/index.html)	How do I navigate and look around the system?

11:10	5. Writing and reading files (./03-files/index.html)	How do I create/edit text files? How do I move/copy/delete files?
11:55	6. Lunch Break (./031-lunch/index.html)	Break
12:55	7. Wildcards and pipes (./04-wildcards-pipes/index.html)	How can I run a command on multiple files at once? Is there an easy way of saving a command's output?
13:50	8. Scripts, variables, and loops (./05-scripts/index.html)	How do I turn a set of commands into a program?
14:45	9. Coffee Break (./051-coffee/index.html)	Break
15:15	Finish	

The actual schedule may vary slightly depending on the topics and exercises chosen by the instructor.

Licensed under CC-BY 4.0 (https://creativecommons.org/licenses/by/4.0/) 2018–2020 by The Carpentries (https://carpentries.org/)
Licensed under CC-BY 4.0 (https://creativecommons.org/licenses/by/4.0/) 2016–2018 by Software Carpentry Foundation (https://software-carpentry.org)

Edit on GitHub (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/edit/gh-pages/index.md) / Contributing (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/blob/gh-pages/CONTRIBUTING.md) / Source (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/) / Cite (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/blob/gh-pages/CITATION) / Contact (mailto:team@carpentries.org)

Using The Carpentries style (https://github.com/carpentries/styles/) version 9.5.3 (https://github.com/carpentries/styles/releases/tag/v9.5.3).

Why Use a Cluster?

Overview

Teaching: 15 min
Exercises: 5 min
Questions

- Why would I be interested in High Performance Computing (HPC)?
- What can I expect to learn from this course?

Objectives

- Be able to describe what an HPC system is
- Identify how an HPC system could benefit you.

Frequently, research problems that use computing can outgrow the desktop or laptop computer where they started:

- A statistics student wants to cross-validate their model. This involves running the model 1000 times – but each run takes an hour. Running on their laptop will take over a month!
- A genomics researcher has been using small datasets of sequence data, but soon will be receiving a new type of sequencing data that is 10 times as large. It's already challenging to open the datasets on their computer – analyzing these larger datasets will probably crash it.
- An engineer is using a fluid dynamics package that has an option to run in parallel. So far, they haven't used this option on their desktop, but in going from 2D to 3D simulations, simulation time has more than tripled and it might be useful to take advantage of that feature.

In all these cases, what is needed is access to more computers that can be used at the same time.

And what do you do?

Talk to your neighbour, office mate or rubber duck (https://rubberduckdebugging.com/) about your research. How does computing help you do your research? How could more computing help you do more or better research?

Doing Analysis or Running Code

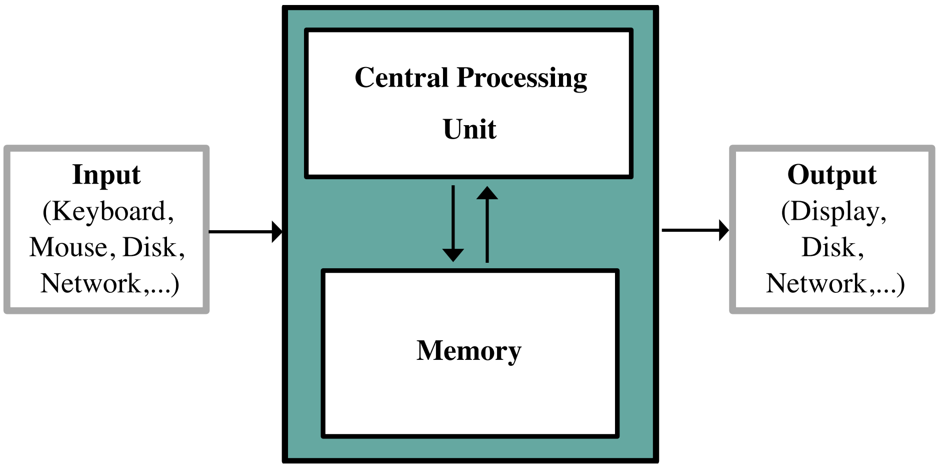
A standard Laptop for standard tasks

Today, people coding or analysing data typically work with laptops.

Let's dissect what resources programs running on a laptop require:

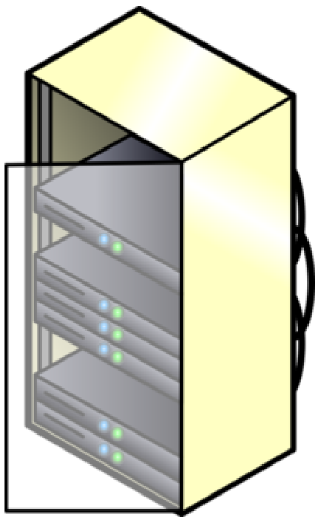
- the keyboard and/or touchpad is used to tell the computer what to do (**Input**)
- the internal computing resources **Central Processing Unit** and **Memory** perform calculation
- the display depicts progress and results (**Output**)

Schematically, this can be reduced to the following:


Central Processing Unit
Input (Keyboard, Mouse, Disk, Network,...)
Memory
Output (Display, Disk, Network,...)

When tasks take too long

When the task to solve become heavy on computations, the operations are typically out-sourced from the local laptop or desktop to elsewhere. Take for example the task to find the directions for your next business trip. The capabilities of your laptop are typically not enough to calculate that route spontaneously. So you use website, which in turn runs on a server that is almost exclusively not in the same room as you are.



Note here, that a server is mostly a noisy computer mounted into a rack cabinet which in turn resides in a data center. The internet made it possible that these data centers do not require to be nearby your laptop. What people call **the cloud** is mostly a web-service where you can rent such servers by providing your credit card details and by clicking together the specs of this remote resource.

The server itself has no direct display or input methods attached to it. But most importantly, it has much more storage, memory and compute capacity than your laptop will ever have. In any case, you need a local device (laptop, workstation, mobile phone or tablet) to interact with this remote machine, which people typically call ‘a server’.

When one server is not enough

If the computational task or analysis to complete is daunting for a single server, larger agglomerations of servers are used. These go by the name of clusters or super computers.

The methodology of providing the input data, communicating options and flags as well as retrieving the results is quite opposite to using a plain laptop. Moreover, using a GUI style interface is often discarded in favor of using the command line. This imposes a double paradigm shift for prospect users:

1. they work with the command line (not a GUI style user interface)
2. they work with a distributed set of computers (called nodes)

I’ve never used a server, have I?

Take a minute and think about which of your daily interactions with a computer may require a remote server or even cluster to provide you with results.

Key Points

- High Performance Computing (HPC) typically involves connecting to very large computing systems elsewhere in the world.
- These other systems can be used to do work that would either be impossible or much slower or smaller systems.
- The standard method of interacting with such systems is via a command line interface called Bash.

(../)

(../01-
conne

Licensed under CC-BY 4.0 (https://creativecommons.org/licenses/by/4.0/) 2018–2020 by The Carpentries (https://carpentries.org/)
Licensed under CC-BY 4.0 (https://creativecommons.org/licenses/by/4.0/) 2016–2018 by Software Carpentry Foundation (https://software-carpentry.org)

Edit on GitHub (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/edit/gh-pages/_episodes/00-hpc-intro.md) / Contributing (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/blob/gh-pages/CONTRIBUTING.md) / Source (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/) / Cite (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/blob/gh-pages/CITATION) / Contact (mailto:team@carpentries.org)

Using The Carpentries style (https://github.com/carpentries/styles/) version 9.5.3 (https://github.com/carpentries/styles/releases/tag/v9.5.3).

Connecting to the remote HPC system

Overview

Teaching: 20 min
Exercises: 10 min
Questions

- How do I open a terminal?
- How do I connect to a remote computer?

Objectives

- Connect to a remote HPC system.

Opening a Terminal

Connecting to an HPC system is most often done through a tool known as "SSH" (Secure SHell) and usually SSH is run through a terminal. So, to begin using an HPC system we need to begin by opening a terminal. Different operating systems have different terminals, none of which are exactly the same in terms of their features and abilities while working on the operating system. However each time you connect to the same remote system with a new terminal the experience will be identical as each will faithfully present the same experience of using that system.

Here is the process for opening a terminal in each operating system.

Linux

There are many different versions (aka "flavours") of Linux and how to open a terminal window can change between flavours. A quick search on the Internet for "how to open a terminal window in" with your particular Linux flavour appended to the end should give you the directions you need.

A very popular version of Linux is Ubuntu. There are many ways to open a terminal window in Ubuntu but a very fast way is to use the terminal shortcut key sequence: Ctrl+Alt+T.

Mac

Macs have had a terminal built in since the first version of OS X (now macOS) as it is built on a UNIX-like operating system, leveraging many parts from BSD (Berkeley Systems Designs). The terminal can be quickly opened through the use of the Searchlight tool. Hold down the command key and press the spacebar. In the search bar that shows up type "terminal", choose the terminal app from the list of results (it will look like a tiny, black computer screen) and you will be presented with a terminal window. Alternatively, you can find Terminal under "Utilities" in the Applications menu.

Windows

If you are using Windows, you should have installed Git Bash as part of the setup for this course (../setup/) which includes an SSH client you can use in the same way as for Linux and Mac. Open the Git Bash program to get terminal access.

Logging onto the system

With all of this in mind, let's connect to a remote HPC system. In this workshop, we will connect to Cirrus — an HPC system located at the EPCC, The University of Edinburgh. Although it's unlikely that every system will be exactly like Cirrus, it's a very good example of what you can expect from an HPC installation. To connect to our example computer, we will use SSH.

SSH allows us to connect to UNIX computers remotely, and use them as if they were our own. The general syntax of the connection command follows the format `ssh yourUsername@some.computer.address` Let's attempt to connect to the HPC system now:

```
ssh yourUsername@login.cirrus.ac.uk
```

```

The authenticity of host 'login.cirrus.ac.uk (129.215.175.28)' can't be established.
ECDSA key fingerprint is SHA256:JRj286Pkqh6aeO5zx1QUkS8un5fpcapmezusceSGhok.
ECDSA key fingerprint is MD5:99:59:db:b1:3f:18:d0:2c:49:4e:c2:74:86:ac:f7:c6.
Are you sure you want to continue connecting (yes/no)?  # type "yes"!
Warning: Permanently added the ECDSA host key for IP address '129.215.175.28' to the list of known hosts.
yourUsername@login.cirrus.ac.uk's password:  # no text appears as you enter your password
Last login: Wed Nov 28 08:46:30 2018 from cpc102380-sgyl38-2-0-cust601.18-2.cable.virginm.net

================================================================================

Cirrus HPC Service

--------------------------------------------------------------------------------
This is a private computing facility. Access to this system is limited to those
who have been granted access by the operating service provider on behalf of the
issuing authority and use is restricted to the purposes for which access was
granted. All access and usage are governed by the terms and conditions of access
agreed to by all registered users and are thus subject to the provisions of the
Computer Misuse Act, 1990 under which unauthorised use is a criminal offence.
--------------------------------------------------------------------------------

For help please contact the Cirrus helpdesk at:
support@cirrus.ac.uk

================================================================================
```

If you've connected successfully, you should see a prompt like the one below. This prompt is informative, and lets you grasp certain information at a glance. (If you don't understand what these things are, don't worry! We will cover things in depth as we explore the system further.)

```
[yourUsername@cirrus-login0 ~]$
```

Telling the Difference between the Local Terminal and the Remote Terminal

You may have noticed that the prompt changed when you logged into the remote system using the terminal (if you logged in using PuTTY this will not apply because it does not offer a local terminal). This change is important because it makes it clear on which system the commands you type will be run when you pass them into the terminal. This change is also a small complication that we will need to navigate throughout the workshop. Exactly what is reported before the `$` in the terminal when it is connected to the local system and the remote system will typically be different for every user. We still need to indicate which system we are entering commands on though so we will adopt the following convention:

- `[local]$` when the command is to be entered on a terminal connected to your local computer
- `[yourUsername@cirrus-login0 ~]$` when the command is to be entered on a terminal connected to the remote system
- `$` when it really doesn't matter which system the terminal is connected to.

Being certain which system your terminal is connected to

If you ever need to be certain which system a terminal you are using is connected to then use the following command: `$ hostname` .

Keep two terminal windows open

It is strongly recommended that you have two terminals open, one connected to the local system and one connected to the remote system, that you can switch back and forth between. If you only use one terminal window then you will need to reconnect to the remote system using one of the methods above when you see a change from `[local]$` to `[yourUsername@cirrus-login0 ~]$` and disconnect when you see the reverse.

Key Points

- To connect to a remote HPC system using SSH: `ssh yourUsername@remote.computer.address`

(../00-hpc-intro/index.html)

(../01
coffe

Licensed under CC-BY 4.0 (https://creativecommons.org/licenses/by/4.0/) 2018–2020 by The Carpentries (https://carpentries.org/)
Licensed under CC-BY 4.0 (https://creativecommons.org/licenses/by/4.0/) 2016–2018 by Software Carpentry Foundation (https://software-carpentry.org)

Edit on GitHub (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/edit/gh-pages/_episodes/01-connecting.md) / Contributing (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/blob/gh-pages/CONTRIBUTING.md) / Source (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/) / Cite (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/blob/gh-pages/CITATION) / Contact (mailto:team@carpentries.org)

Using The Carpentries style (https://github.com/carpentries/styles/) version 9.5.3 (https://github.com/carpentries/styles/releases/tag/v9.5.3).

Moving around and looking at things

Overview

Teaching: 15 min
Exercises: 5 min
Questions

- How do I navigate and look around the system?

Objectives

- Learn how to navigate around directories and look at their contents
- Explain the difference between a file and a directory.
- Translate an absolute path into a relative path and vice versa.
- Identify the actual command, flags, and filenames in a command-line call.
- Demonstrate the use of tab completion, and explain its advantages.

At the point in this lesson, we've just logged into the system. Nothing has happened yet, and we're not going to be able to do anything until we learn a few basic commands. By the end of this lesson, you will know how to "move around" the system and look at what's there.

Right now, all we see is something that looks like this:

```
[yourUsername@cirrus-login0 ~]$
```

The dollar sign is a **prompt**, which shows us that the shell is waiting for input; your shell may use a different character as a prompt and may add information before the prompt. When typing commands, either from these lessons or from other sources, do not type the prompt, only the commands that follow it.

Type the command `whoami` , then press the Enter key (sometimes marked Return) to send the command to the shell. The command's output is the ID of the current user, i.e., it shows us who the shell thinks we are:

```
$ whoami
```

```
yourUsername
```

More specifically, when we type `whoami` the shell:

1. finds a program called `whoami` ,
2. runs that program,
3. displays that program's output, then
4. displays a new prompt to tell us that it's ready for more commands.

Next, let's find out where we are by running a command called `pwd` (which stands for "print working directory"). At any moment, our **current working directory** (where we are) is the directory that the computer assumes we want to run commands in unless we explicitly specify something else. Here, the computer's response is `/lustre/home/tc008/yourUsername` , which is `yourUsername` **home directory**. Note that the location of your home directory may differ from system to system.

```
$ pwd
```

```
/lustre/home/tc008/yourUsername
```

So, we know where we are. How do we look and see what's in our current directory?

```
$ ls
```

`ls` prints the names of the files and directories in the current directory in alphabetical order, arranged neatly into columns.

Differences between remote and local system

Open a second terminal window on your local computer and run the `ls` command without logging in remotely. What differences do you see?

Solution

You would likely see something more like this:

```
Applications Documents    Library      Music        Public
Desktop      Downloads    Movies       Pictures
```

In addition you should also note that the preamble before the prompt (`$`) is different. This is very important for making sure you know what system you are issuing commands on when in the shell.

If nothing shows up when you run `ls`, it means that nothing's there. Let's make a directory for us to play with.

`mkdir <new directory name>` makes a new directory with that name in your current location. Notice that this command required two pieces of input: the actual name of the command (`mkdir`) and an argument that specifies the name of the directory you wish to create.

```
$ mkdir documents
```

Let's us `ls` again. What do we see?

Our folder is there, awesome. What if we wanted to go inside it and do stuff there? We will use the `cd` (change directory) command to move around. Let's `cd` into our new documents folder.

```
$ cd documents
$ pwd
```

```
/lustre/home/tc008/yourUserName/documents
```

Now that we know how to use `cd`, we can go anywhere. That's a lot of responsibility. What happens if we get "lost" and want to get back to where we started?

To go back to your home directory, the following two commands will work:

```
$ cd /lustre/home/tc008/yourUserName
$ cd ~
```

What is the `~` character? When using the shell, `~` is a shortcut that represents `/lustre/home/tc008/yourUserName`.

A quick note on the structure of a UNIX (Linux/Mac/Android/Solaris/etc) filesystem. Directories and absolute paths (i.e. exact position in the system) are always prefixed with a `/`. `/` is the "root" or base directory.

Let's go there now, look around, and then return to our home directory.

```
$ cd /
$ ls
$ cd ~
```

```
bin   cvmfs  etc   initrd  lib64  localscratch  mnt  opt   project  root  sbin     srv  tmp  var
boot  dev    home  lib     local  media         nix  proc  ram      run   scratch  sys  usr  work
```

The “home” directory is the one where we generally want to keep all of our files. Other folders on a UNIX OS contain system files, and get modified and changed as you install new software or upgrade your OS.

📌 Using HPC filesystems

On HPC systems, you have a number of places where you can store your files. These differ in both the amount of space allocated and whether or not they are backed up.

File storage locations:

- **Network filesystem** - Your home directory is an example of a network filesystem. Data stored here is available throughout the HPC system and files stored here are often backed up (but check you local configuration to be sure!). Files stored here are typically slower to access, the data is actually stored on another computer and is being transmitted and made available over the network!
- **Scratch** - Some systems may offer “scratch” space. Scratch space is typically faster to use than your home directory or network filesystem, but is not usually backed up, and should not be used for long term storage.
- **Work file system** - As an alternative to (or sometimes as well as) Scratch space, some HPC systems offer fast file system access as a work file system. Typically, this will have higher performance than your home directory or network file system and may not be backed up. It differs from scratch space in that files in a work file system are not automatically deleted for you, you must manage the space yourself.
- **Local scratch (job only)** - Some systems may offer local scratch space while executing a job. Such storage is very fast, but will be deleted at the end of your job.
- **Ramdisk (job only)** - Some systems may let you store files in a “RAM disk” while running a job, where files are stored directly in the computer’s memory. This extremely fast, but files stored here will count against your job’s memory usage and be deleted at the end of your job.

There are several other useful shortcuts you should be aware of.

- `.` represents your current directory
- `..` represents the “parent” directory of your current location
- While typing nearly *anything*, you can have bash try to autocomplete what you are typing by pressing the `tab` key.

Let’s try these out now:

```
$ cd ./documents
$ pwd
$ cd ..
$ pwd
```

```
/lustre/home/tc008/yourUserName/documents
/lustre/home/tc008/yourUserName
```

Many commands also have multiple behaviours that you can invoke with command line ‘flags.’ What is a flag? It’s generally just your command followed by a ‘-‘ and the name of the flag (sometimes it’s ‘–’ followed by the name of the flag. You follow the flag(s) with any additional arguments you might need.

We’re going to demonstrate a couple of these “flags” using `ls`.

Show hidden files with `-a`. Hidden files are files that begin with `.`, these files will not appear otherwise, but that doesn’t mean they aren’t there! “Hidden” files are not hidden for security purposes, they are usually just config files and other tempfiles that the user doesn’t necessarily need to see all the time.

```
$ ls -a
```

```
.  ..  .bash_logout  .bash_profile  .bashrc  documents  .emacs  .mozilla  .ssh
```

Notice how both `.` and `..` are visible as hidden files. Show files, their size in bytes, date last modified, permissions, and other things with `-l`.

```
$ ls -l
```

```
drwxr-xr-x 2 yourUsername tc001 4096 Jan 14 17:31 documents
```

This is a lot of information to take in at once, but we will explain this later! `ls -l` is *extremely* useful, and tells you almost everything you need to know about your files without actually looking at them.

We can also use multiple flags at the same time!

```
$ ls -l -a
```

```
[yourUsername@cirrus-login0 ~]$  ls -la
total 36
drwx--S--- 5 yourUsername tc001 4096 Nov 28 09:58 .
drwxr-x--- 3 root         tc001 4096 Nov 28 09:40 ..
-rw-r--r-- 1 yourUsername tc001   18 Dec  6  2016 .bash_logout
-rw-r--r-- 1 yourUsername tc001  193 Dec  6  2016 .bash_profile
-rw-r--r-- 1 yourUsername tc001  231 Dec  6  2016 .bashrc
drwxr-sr-x 2 yourUsername tc001 4096 Nov 28 09:58 documents
-rw-r--r-- 1 yourUsername tc001  334 Mar  3  2017 .emacs
drwxr-xr-x 4 yourUsername tc001 4096 Aug  2  2016 .mozilla
drwx--S--- 2 yourUsername tc001 4096 Nov 28 09:58 .ssh
```

Flags generally precede any arguments passed to a UNIX command. `ls` actually takes an extra argument that specifies a directory to look into. When you use flags and arguments together, the syntax (how it's supposed to be typed) generally looks something like this:

```
$ command <flags/options> <arguments>
```

So using `ls -l -a` on a different directory than the one we're in would look something like:

```
$ ls -l -a ~/documents
```

```
drwxr-sr-x 2 yourUsername tc001 4096 Nov 28 09:58 .
drwx--S--- 5 yourUsername tc001 4096 Nov 28 09:58 ..
```

Where to go for help?

How did I know about the `-l` and `-a` options? Is there a manual we can look at for help when we need help? There is a very helpful manual for most UNIX commands: `man` (if you've ever heard of a "man page" for something, this is what it is).

```
$ man ls
```

```
LS(1)                                                 User Commands
LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information  about the FILEs (the current directory by default).  Sort entries alphabeti
cally if none of -cftu-
       vSUX nor --sort is specified.

       Mandatory arguments to long options are mandatory for short options too.
Manual page ls(1) line 1 (press h for help or q to quit)
```

To navigate through the `man` pages, you may use the up and down arrow keys to move line-by-line, or try the spacebar and “b” keys to skip up and down by full page. Quit the `man` pages by typing “q”.

Alternatively, most commands you run will have a `--help` option that displays addition information For instance, with `ls` :

```
$ ls --help
```

```
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

Mandatory arguments to long options are mandatory for short options too.
  -a, --all                  do not ignore entries starting with .
  -A, --almost-all           do not list implied . and ..
      --author               with -l, print the author of each file
  -b, --escape               print C-style escapes for nongraphic characters
      --block-size=SIZE      scale sizes by SIZE before printing them; e.g.,
                               '--block-size=M' prints sizes in units of
                               1,048,576 bytes; see SIZE format below
  -B, --ignore-backups       do not list implied entries ending with ~

# further output omitted for clarity
```

Unsupported command-line options

If you try to use an option that is not supported, `ls` and other programs will print an error message similar to this:

```
[remote]$ ls -j
```

```
ls: invalid option -- 'j'
Try 'ls --help' for more information.
```

Looking at documentation

Looking at the man page for `ls` or using `ls --help` , what does the command `ls` do when used with the `-l` and `-h` arguments?

Some of its output is about properties that we do not cover in this lesson (such as file permissions and ownership), but the rest should be useful nevertheless.

Solution

The `-l` arguments makes `ls` use a **l**ong listing format, showing not only the file/directory names but also additional information such as the file size and the time of its last modification. The `-h` argument makes the file size “**h**uman readable”, i.e. display something like `5.3K` instead of `5369` .

Absolute vs Relative Paths

Starting from `/Users/amanda/data/` , which of the following commands could Amanda use to navigate to her home directory, which is `/Users/amanda` ?

1. `cd .`
2. `cd /`
3. `cd /home/amanda`
4. `cd ../..`
5. `cd ~`
6. `cd home`
7. `cd ~/data/..`
8. `cd`
9. `cd ..`

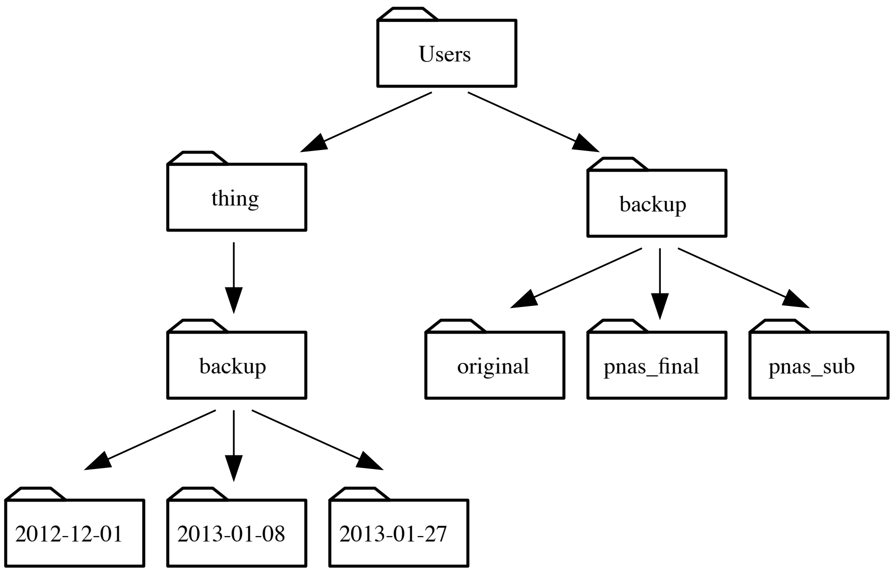
Solution

1. No: `.` stands for the current directory.
2. No: `/` stands for the root directory.
3. No: Amanda's home directory is `/Users/amanda` .
4. No: this goes up two levels, i.e. ends in `/Users` .
5. Yes: `~` stands for the user's home directory, in this case `/Users/amanda` .
6. No: this would navigate into a directory `home` in the current directory if it exists.
7. Yes: unnecessarily complicated, but correct.
8. Yes: shortcut to go back to the user's home directory.
9. Yes: goes up one level.

Relative Path Resolution

Using the filesystem diagram below, if `pwd` displays `/Users/thing` , what will `ls -F ../backup` display?

1. `../backup: No such file or directory`
2. `2012-12-01 2013-01-08 2013-01-27`
3. `2012-12-01/ 2013-01-08/ 2013-01-27/`
4. `original/ pnas_final/ pnas_sub/`



Solution

1. No: there *is* a directory `backup` in `/Users` .
2. No: this is the content of `Users/thing/backup` , but with `..` we asked for one level further up.
3. No: see previous explanation.
4. Yes: `../backup/` refers to `/Users/backup/` .

`ls` Reading Comprehension

Assuming a directory structure as in the above Figure (File System for Challenge Questions), if `pwd` displays `/Users/backup` , and `-r` tells `ls` to display things in reverse order, what command will display:

```
pnas_sub/ pnas_final/ original/
```

1. `ls pwd`
2. `ls -r -F`
3. `ls -r -F /Users/backup`

Solution

1. No: `pwd` is not the name of a directory.
2. Yes: `ls` without directory argument lists files and directories in the current directory.
3. Yes: uses the absolute path explicitly.

Key Points

- Your current directory is referred to as the working directory.
- To change directories, use `cd` .
- To view files, use `ls` .
- You can view help for a command with `man command` or `command --help` .
- Hit `tab` to autocomplete whatever you're currently typing.

(../011-coffee/index.html)

(../03
files/i

Licensed under CC-BY 4.0 (https://creativecommons.org/licenses/by/4.0/) 2018–2020 by The Carpentries (https://carpentries.org/)
Licensed under CC-BY 4.0 (https://creativecommons.org/licenses/by/4.0/) 2016–2018 by Software Carpentry Foundation (https://software-carpentry.org)

Edit on GitHub (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/edit/gh-pages/_episodes/02-navigation.md) / Contributing (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/blob/gh-pages/CONTRIBUTING.md) / Source (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/) / Cite (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/blob/gh-pages/CITATION) / Contact (mailto:team@carpentries.org)

Using The Carpentries style (https://github.com/carpentries/styles/) version 9.5.3 (https://github.com/carpentries/styles/releases/tag/v9.5.3).

Writing and reading files

Overview

Teaching: 30 min
Exercises: 15 min
Questions

- How do I create/edit text files?
- How do I move/copy/delete files?

Objectives

- Learn to use the `nano` text editor.
- Understand how to move, create, and delete files.

Now that we know how to move around and look at things, let's learn how to read, write, and handle files! We'll start by moving back to our home directory and creating a scratch directory:

```
$ cd ~
$ mkdir hpc-test
$ cd hpc-test
```

Creating and Editing Text Files

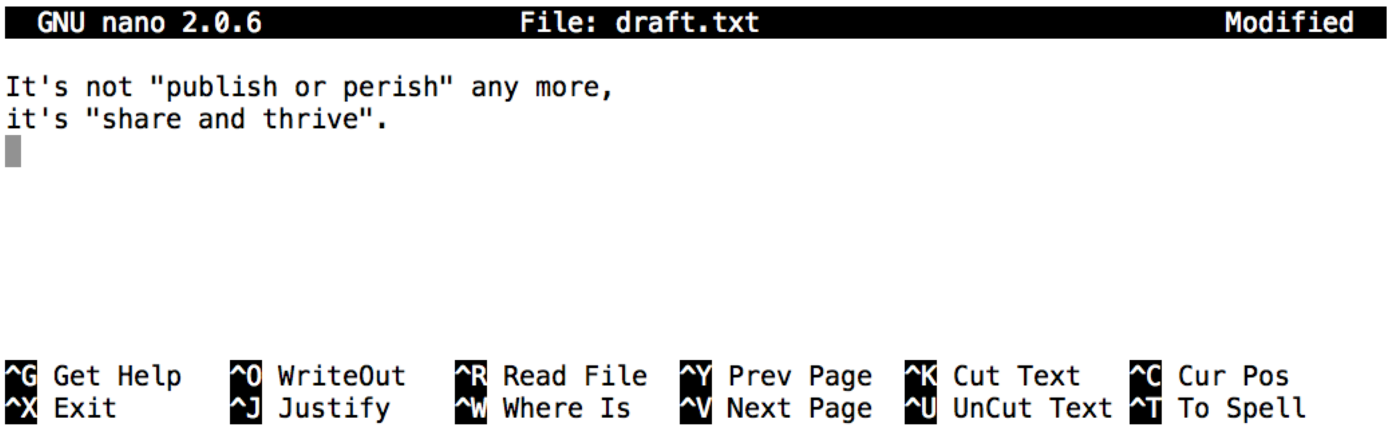
When working on an HPC system, we will frequently need to create or edit text files. Text is one of the simplest computer file formats, defined as a simple sequence of text lines.

What if we want to make a file? There are a few ways of doing this, the easiest of which is simply using a text editor. For this lesson, we are going to us `nano`, since it's more intuitive than many other terminal text editors.

`nano` is another command like `ls`, but one that comes in a 'module'. We will talk more about modules later, but for now type `module load nano` to make the command available.

Now, to create or edit a file, type `nano <filename>`, on the terminal, where `<filename>` is the name of the file. If the file does not already exist, it will be created. Let's make a new file now, type whatever you want in it, and save it.

```
$ nano draft.txt
```



```
  GNU nano 2.0.6                  File: draft.txt                          Modified

It's not "publish or perish" any more,
it's "share and thrive".



^G Get Help   ^O WriteOut   ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is   ^V Next Page  ^U UnCut Text  ^T To Spell
```

Nano defines a number of *shortcut keys* (prefixed by the Control or Ctrl key) to perform actions such as saving the file or exiting the editor. Here are the shortcut keys for a few common actions:

- Ctrl + O — save the file (into a current name or a new name).
- Ctrl + X — exit the editor. If you have not saved your file upon exiting, `nano` will ask you if you want to save.
- Ctrl + K — cut ("kill") a text line. This command deletes a line and saves it on a clipboard. If repeated multiple times without any interruption (key typing or cursor movement), it will cut a chunk of text lines.
- Ctrl + U — paste the cut text line (or lines). This command can be repeated to paste the same text elsewhere.

Using `vim` as a text editor

From time to time, you may encounter the `vim` text editor. Although `vim` isn't the easiest or most user-friendly of text editors, you'll be able to find it on any system and it has many more features than `nano` .

`vim` has several modes, a "command" mode (for doing big operations, like saving and quitting) and an "insert" mode. You can switch to insert mode with the `i` key, and command mode with `Esc` .

In insert mode, you can type more or less normally. In command mode there are a few commands you should be aware of:

- `:q!` — quit, without saving
- `:wq` — save and quit
- `dd` — cut/delete a line
- `y` — paste a line

Do a quick check to confirm our file was created.

```
$ ls
```

```
draft.txt
```

Reading Files

Let's read the file we just created now. There are a few different ways of doing this, one of which is reading the entire file with `cat` .

```
$ cat draft.txt
```

```
It's not "publish or perish" any more,
it's "share and thrive".
```

By default, `cat` prints out the content of the given file. Although `cat` may not seem like an intuitive command with which to read files, it stands for "concatenate". Giving it multiple file names will print out the contents of the input files in the order specified in the `cat` 's invocation. For example,

```
$ cat draft.txt draft.txt
```

```
It's not "publish or perish" any more,
it's "share and thrive".
It's not "publish or perish" any more,
it's "share and thrive".
```

Reading Multiple Text Files

Create two more files using `nano` , giving them different names such as `chap1.txt` and `chap2.txt` . Then use a single `cat` command to read and print the contents of `draft.txt` , `chap1.txt` , and `chap2.txt` .

Creating Directory

We've successfully created a file. What about a directory? We've actually done this before, using `mkdir`.

```
$ mkdir files
$ ls
```

```
draft.txt  files
```

Moving, Renaming, Copying Files

Moving—We will move `draft.txt` to the `files` directory with `mv` ("move") command. The same syntax works for both files and directories: `mv <file/directory> <new-location>`

```
$ mv draft.txt files
$ cd files
$ ls
```

```
draft.txt
```

Renaming— `draft.txt` isn't a very descriptive name. How do we go about changing it? It turns out that `mv` is also used to rename files and directories. Although this may not seem intuitive at first, think of it as *moving* a file to be stored under a different name. The syntax is quite similar to moving files: `mv oldName newName`.

```
$ mv draft.txt newname.testfile
$ ls
```

```
newname.testfile
```

File extensions are arbitrary

In the last example, we changed both a file's name and extension at the same time. On UNIX systems, file extensions (like `.txt`) are arbitrary. A file is a `.txt` file only because we say it is. Changing the name or extension of the file will *never* change a file's contents, so you are free to rename things as you wish. With that in mind, however, file extensions are a useful tool for keeping track of what type of data it contains. A `.txt` file typically contains text, for instance.

Copying—What if we want to copy a file, instead of simply renaming or moving it? Use `cp` command (an abbreviated name for "copy"). This command has two different uses that work in the same way as `mv`:

- Copy to same directory (copied file is renamed): `cp file newFilename`
- Copy to other directory (copied file retains original name): `cp file directory`

Let's try this out.

```
$ cp newname.testfile copy.testfile
$ ls
$ cp newname.testfile ..
$ cd ..
$ ls
```

```
newname.testfile copy.testfile
files documents newname.testfile
```

Removing files

We've begun to clutter up our workspace with all of the directories and files we've been making. Let's learn how to get rid of them. One important note before we start… **when you delete a file on UNIX systems, they are gone *forever***. There is no "recycle bin" or "trash". Once a file is deleted, it is gone, never to return. So be *very* careful when deleting files.

Files are deleted with `rm file [moreFiles]` . To delete the `newname.testfile` in our current directory:

```
$ ls
$ rm newname.testfile
$ ls
```

```
files Documents newname.testfile
files Documents
```

That was simple enough. Directories are deleted in a similar manner using `rm -r` (the `-r` option stands for 'recursive').

```
$ ls
$ rm -r Documents
$ rm -r files
$ ls
```

`rm -r directory` is probably the scariest command on UNIX- it will delete a directory and all of its contents. **ALWAYS** double check your typing before using it.

What happens when you use `rm -rf` accidentally

Steam is a major online sales platform for PC video games with over 125 million users. Despite this, it hasn't always had the most stable or error-free code.

In January 2015, user kevyin on GitHub reported that Steam's Linux client had deleted every file on his computer (https://github.com/ValveSoftware/steam-for-linux/issues/3671). It turned out that one of the Steam programmers had added the following line: `rm -rf "$STEAMROOT/"*` (The `-f` option is used to force delete write-protected files and silence warnings). Due to the way that Steam was set up, the variable `$STEAMROOT` was never initialized, meaning the statement evaluated to `rm -rf /*`. This coding error in the Linux client meant that Steam deleted every single file on a computer when run in certain scenarios (including connected external hard drives). Moral of the story: **be very careful** when using `rm -rf` !

Looking at files

Sometimes it's not practical to read an entire file with `cat` - the file might be way too large, take a long time to open, or maybe we want to only look at a certain part of the file. As an example, we are going to look at a large and complex file type used in bioinformatics- a .gtf file. The GTF2 format is commonly used to describe the location of genetic features in a genome.

Let's grab and unpack a set of demo files for use later. To do this, we'll use `wget` (`wget link` downloads a file from a link).

```
$ wget https://aniabrown.github.io/hpc-carpentry-shell-WHPC/files/bash-lesson.tar.gz
```

You'll commonly encounter `.tar.gz` archives while working in UNIX. To extract the files from a `.tar.gz` file, we run the command `tar -xvf filename.tar.gz` :

```
$ tar -xvf bash-lesson.tar.gz
```

```
dmel-all-r6.19.gtf
dmel_unique_protein_isoforms_fb_2016_01.tsv
gene_association.fb
SRR307023_1.fastq
SRR307023_2.fastq
SRR307024_1.fastq
SRR307024_2.fastq
SRR307025_1.fastq
SRR307025_2.fastq
SRR307026_1.fastq
SRR307026_2.fastq
SRR307027_1.fastq
SRR307027_2.fastq
SRR307028_1.fastq
SRR307028_2.fastq
SRR307029_1.fastq
SRR307029_2.fastq
SRR307030_1.fastq
SRR307030_2.fastq
```

Unzipping files

We just unzipped a .tar.gz file for this example. What if we run into other file formats that we need to unzip? Just use the handy reference below:

- `gunzip` extracts the contents of .gz files
- `unzip` extracts the contents of .zip files
- `tar -xvf` extracts the contents of .tar.gz and .tar.bz2 files

That is a lot of files! One of these files, `dmel-all-r6.19.gtf` is extremely large, and contains every annotated feature in the *Drosophila melanogaster* genome. It's a huge file- what happens if we run `cat` on it? (Press `Ctrl + C` to stop it).

So, `cat` is a really bad option when reading big files… it scrolls through the entire file far too quickly! What are the alternatives? Try all of these out and see which ones you like best!

- `head file` : Print the top 10 lines in a file to the console. You can control the number of lines you see with the `-n numberOfLines` flag.
- `tail file` : Same as `head` , but prints the last 10 lines in a file to the console.
- `less file` : Opens a file and display as much as possible on-screen. You can scroll with `Enter` or the arrow keys on your keyboard. Press `q` to close the viewer.

Out of `cat` , `head` , `tail, and` less`, which method of reading files is your favourite? Why?

Key Points

- Use `nano` to create or edit text files from a terminal.
- `cat file1 [file2 ...]` prints the contents of one or more files to terminal.
- `mv old dir` moves a file or directory to another directory `dir` .
- `mv old new` renames a file or directory.
- `cp old new` copies a file.
- `cp old dir` copies a file to another directory `dir` .
- `rm path` deletes (removes) a file.
- File extensions are entirely arbitrary on UNIX systems.

‹ (../02-navigation/index.html)

› (../03
lunch

Licensed under CC-BY 4.0 (https://creativecommons.org/licenses/by/4.0/) 2018–2020 by The Carpentries (https://carpentries.org/)
Licensed under CC-BY 4.0 (https://creativecommons.org/licenses/by/4.0/) 2016–2018 by Software Carpentry Foundation (https://software-carpentry.org)

Edit on GitHub (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/edit/gh-pages/_episodes/03-files.md) / Contributing (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/blob/gh-pages/CONTRIBUTING.md) / Source (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/) / Cite (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/blob/gh-pages/CITATION) / Contact (mailto:team@carpentries.org)

Using The Carpentries style (https://github.com/carpentries/styles/) version 9.5.3 (https://github.com/carpentries/styles/releases/tag/v9.5.3).

Wildcards and pipes

Overview

Teaching: 45 min
Exercises: 10 min
Questions

- How can I run a command on multiple files at once?
- Is there an easy way of saving a command's output?

Objectives

- Redirect a command's output to a file.
- Process a file instead of keyboard input using redirection.
- Construct command pipelines with two or more stages.
- Explain what usually happens if a program or pipeline isn't given any input to process.

Required files

If you didn't get them in the last lesson, make sure to download the example files used in the next few sections:

Using wget — `wget https://aniabrown.github.io/hpc-carpentry-shell-WHPC/files/bash-lesson.tar.gz`

Using a web browser — https://aniabrown.github.io/hpc-carpentry-shell-WHPC/files/bash-lesson.tar.gz (https://aniabrown.github.io/hpc-carpentry-shell-WHPC/files/bash-lesson.tar.gz)

Now that we know some of the basic UNIX commands, we are going to explore some more advanced features. The first of these features is the wildcard `*`. In our examples before, we've done things to files one at a time and otherwise had to specify things explicitly. The `*` character lets us speed things up and do things across multiple files.

Ever wanted to move, delete, or just do "something" to all files of a certain type in a directory? `*` lets you do that, by taking the place of one or more characters in a piece of text. So `*.txt` would be equivalent to all `.txt` files in a directory for instance. `*` by itself means all files. Let's use our example data to see what I mean.

```
$ ls
```

```
bash-lesson.tar.gz                             SRR307024_2.fastq  SRR307028_1.fastq
dmel-all-r6.19.gtf                             SRR307025_1.fastq  SRR307028_2.fastq
dmel_unique_protein_isoforms_fb_2016_01.tsv    SRR307025_2.fastq  SRR307029_1.fastq
gene_association.fb                            SRR307026_1.fastq  SRR307029_2.fastq
SRR307023_1.fastq                              SRR307026_2.fastq  SRR307030_1.fastq
SRR307023_2.fastq                              SRR307027_1.fastq  SRR307030_2.fastq
SRR307024_1.fastq                              SRR307027_2.fastq
```

Now we have a whole bunch of example files in our directory. For this example we are going to learn a new command that tells us how long a file is: `wc`. `wc -l file` tells us the length of a file in lines.

```
$ wc -l dmel-all-r6.19.gtf
```

```
542048 dmel-all-r6.19.gtf
```

Interesting, there are over 540000 lines in our `dmel-all-r6.19.gtf` file. What if we wanted to run `wc -l` on every .fastq file? This is where `*` comes in really handy! `*.fastq` would match every file ending in `.fastq`.

```
$ wc -l *.fastq
```

```
20000 SRR307023_1.fastq
20000 SRR307023_2.fastq
20000 SRR307024_1.fastq
20000 SRR307024_2.fastq
20000 SRR307025_1.fastq
20000 SRR307025_2.fastq
20000 SRR307026_1.fastq
20000 SRR307026_2.fastq
20000 SRR307027_1.fastq
20000 SRR307027_2.fastq
20000 SRR307028_1.fastq
20000 SRR307028_2.fastq
20000 SRR307029_1.fastq
20000 SRR307029_2.fastq
20000 SRR307030_1.fastq
20000 SRR307030_2.fastq
320000 total
```

That was easy. What if we wanted to do the same command, except on every file in the directory? A nice trick to keep in mind is that `*` by itself matches *every* file.

```
$ wc -l *
```

```
   53037 bash-lesson.tar.gz
  542048 dmel-all-r6.19.gtf
   22129 dmel_unique_protein_isoforms_fb_2016_01.tsv
  106290 gene_association.fb
   20000 SRR307023_1.fastq
   20000 SRR307023_2.fastq
   20000 SRR307024_1.fastq
   20000 SRR307024_2.fastq
   20000 SRR307025_1.fastq
   20000 SRR307025_2.fastq
   20000 SRR307026_1.fastq
   20000 SRR307026_2.fastq
   20000 SRR307027_1.fastq
   20000 SRR307027_2.fastq
   20000 SRR307028_1.fastq
   20000 SRR307028_2.fastq
   20000 SRR307029_1.fastq
   20000 SRR307029_2.fastq
   20000 SRR307030_1.fastq
   20000 SRR307030_2.fastq
 1043504 total
```

Multiple wildcards

You can even use multiple `*` s at a time. How would you run `wc -l` on every file with "fb" in it?

Solution

```
wc -l *fb*
```

i.e. *anything or nothing* then `fb` then *anything or nothing*

Using other commands

Now let's try cleaning up our working directory a bit. Create a folder called "fastq" and move all of our .fastq files there in one `mv` command.

Solution

```
mkdir fastq
mv *.fastq fastq/
```

Redirecting output

Each of the commands we've used so far does only a very small amount of work. However, we can chain these small UNIX commands together to perform otherwise complicated actions!

For our first foray into *piping*, or redirecting output, we are going to use the `>` operator to write output to a file. When using `>`, whatever is on the left of the `>` is written to the filename you specify on the right of the arrow. The actual syntax looks like `command > filename`.

Let's try several basic usages of `>`. `echo` simply prints back, or echoes whatever you type after it.

```
$ echo "this is a test"
$ echo "this is a test" > test.txt
$ ls
$ cat test.txt
```

```
this is a test

bash-lesson.tar.gz                            fastq
dmel-all-r6.19.gtf                            gene_association.fb
dmel_unique_protein_isoforms_fb_2016_01.tsv  test.txt

this is a test
```

Awesome, let's try that with a more complicated command, like `wc -l`.

```
$ wc -l * > word_counts.txt
$ cat word_counts.txt
```

```
wc: fastq: Is a directory

    53037 bash-lesson.tar.gz
   542048 dmel-all-r6.19.gtf
    22129 dmel_unique_protein_isoforms_fb_2016_01.tsv
        0 fastq
   106290 gene_association.fb
        1 test.txt
   723505 total
```

Notice how we still got some output to the console even though we "piped" the output to a file? Our expected output still went to the file, but how did the error message get skipped and not go to the file?

This phenomena is an artefact of how UNIX systems are built. There are 3 input/output streams for every UNIX program you will run: `stdin`, `stdout`, and `stderr`.

Let's dissect these three streams of input/output in the command we just ran: `wc -l * > word_counts.txt`

- `stdin` is the input to a program. In the command we just ran, `stdin` is represented by `*`, which is simply every filename in our current directory.

- `stdout` contains the actual, expected output. In this case, `>` redirected `stdout` to the file `word_counts.txt`.
- `stderr` typically contains error messages and other information that doesn't quite fit into the category of "output". If we insist on redirecting both `stdout` and `stderr` to the same file we would use `&>` instead of `>`. (We can redirect just `stderr` using `2>`.)

Knowing what we know now, let's try re-running the command, and send all of the output (including the error message) to the same `word_counts.txt` files as before.

```
$ wc -l * &> word_counts.txt
```

Notice how there was no output to the console that time. Let's check that the error message went to the file like we specified.

```
$ cat word_counts.txt
```

```
    53037 bash-lesson.tar.gz
   542048 dmel-all-r6.19.gtf
    22129 dmel_unique_protein_isoforms_fb_2016_01.tsv
wc: fastq: Is a directory
        0 fastq
   106290 gene_association.fb
        1 test.txt
        7 word_counts.txt
   723512 total
```

Success! The `wc: fastq: Is a directory` error message was written to the file. Also, note how the file was silently overwritten by directing output to the same place as before. Sometimes this is not the behaviour we want. How do we append (add) to a file instead of overwriting it?

Appending to a file is done the same was as redirecting output. However, instead of `>`, we will use `>>`.

```
$ echo "We want to add this sentence to the end of our file" >> word_counts.txt
$ cat word_counts.txt
```

```
  22129 dmel_unique_protein_isoforms_fb_2016_01.tsv
 471308 Drosophila_melanogaster.BDGP5.77.gtf
      0 fastq
1304914 fb_synonym_fb_2016_01.tsv
 106290 gene_association.fb
      1 test.txt
1904642 total
We want to add this sentence to the end of our file
```

Chaining commands together

We now know how to redirect `stdout` and `stderr` to files. We can actually take this a step further and redirect output (`stdout`) from one command to serve as the input (`stdin`) for the next. To do this, we use the `|` (pipe) operator.

`grep` is an extremely useful command. It finds things for us within files. Basic usage (there are a lot of options for more clever things, see the `man` page) uses the syntax `grep whatToFind fileToSearch`. Let's use `grep` to find all of the entries pertaining to the `Act5C` gene in *Drosophila melanogaster*.

```
$ grep Act5C dmel-all-r6.19.gtf
```

The output is nearly unintelligible since there is so much of it. Let's send the output of that `grep` command to `head` so we can just take a peek at the first line. The `|` operator lets us send output from one command to the next:

```
$ grep Act5C dmel-all-r6.19.gtf | head -n 1
```

```
X       FlyBase gene    5900861 5905399 .       +       .       gene_id "FBgn0000042"; gene_symbol "Act5
C";
```

Nice work, we sent the output of `grep` to `head` . Let's try counting the number of entries for Act5C with `wc -l` . We can do the same trick to send `grep` 's output to `wc -l` :

```
$ grep Act5C dmel-all-r6.19.gtf | wc -l
```

```
46
```

Note that this is just the same as redirecting output to a file, then reading the number of lines from that file.

Writing commands using pipes

How many files are there in the "fastq" directory we made earlier? (Use the shell to do this.)

Solution

```
ls fastq/ | wc -l
```

Output of `ls` is one line per item so counting lines gives the number of files.

Reading from compressed files

Let's compress one of our files using gzip.

```
$ gzip gene_association.fb
```

`zcat` acts like `cat` , except that it can read information from `.gz` (compressed) files. Using `zcat` , can you write a command to take a look at the top few lines of the `gene_association.fb.gz` file (without decompressing the file itself)?

Solution

```
zcat gene_association.fb.gz | head
```

The `head` command without any options shows the first 10 lines of a file

Key Points

- The `*` wildcard is used as a placeholder to match any text that follows a pattern.
- Redirect a commands output to a file with `>` .
- Commands can be chained with `|`

(../031-lunch/index.html)

(../05-script

Licensed under CC-BY 4.0 (https://creativecommons.org/licenses/by/4.0/) 2018–2020 by The Carpentries (https://carpentries.org/)
Licensed under CC-BY 4.0 (https://creativecommons.org/licenses/by/4.0/) 2016–2018 by Software Carpentry Foundation (https://software-carpentry.org)

Edit on GitHub (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/edit/gh-pages/_episodes/04-wildcards-pipes.md) / Contributing (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/blob/gh-pages/CONTRIBUTING.md) / Source (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/) / Cite (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/blob/gh-pages/CITATION) / Contact (mailto:team@carpentries.org)

Using The Carpentries style (https://github.com/carpentries/styles/) version 9.5.3 (https://github.com/carpentries/styles/releases/tag/v9.5.3).

Scripts, variables, and loops

Overview

Teaching: 45 min
Exercises: 10 min
Questions

- How do I turn a set of commands into a program?

Objectives

- Write a shell script
- Understand shell variables and how to use them
- Write a simple for loop
- Understand and manipulate UNIX permissions

We now know a lot of UNIX commands! Wouldn't it be great if we could save certain commands so that we could run them later or not have to type them out again? As it turns out, this is straightforward to do. A "shell script" is essentially a text file containing a list of UNIX commands to be executed in a sequential manner. These shell scripts can be run whenever we want, and are a great way to automate our work.

Writing a Script

So how do we write a shell script, exactly? It turns out we can do this with a text editor. Start editing a file called "demo.sh" (to recap, we can do this with `nano demo.sh`). The ".sh" is the standard file extension for shell scripts that most people use (you may also see ".bash" used).

Our shell script will have two parts:

- On the very first line, add `#!/bin/bash`. The `#!` (pronounced "hash-bang") tells our computer what program to run our script with. In this case, we are telling it to run our script with our command-line shell (what we've been doing everything in so far). If we wanted our script to be run with something else, like Perl, we could add `#!/usr/bin/perl`
- Now, anywhere below the first line, add `echo "Our script worked!"`. When our script runs, `echo` will happily print out `Our script worked!`.

Our file should now look like this:

```
#!/bin/bash

echo "Our script worked!"
```

Ready to run our program? Let's try running it:

```
$ bash demo.sh
```

Fantastic, we've written our first program! Before we go any further, let's learn how to take notes inside our program using comments. A comment is indicated by the `#` character, followed by whatever we want. Comments do not get run. Let's try out some comments in the console, then add one to our script!

```
# This wont show anything
```

Now lets try adding this to our script with `nano`. Edit your script to look something like this:

```
#!/bin/bash

# This is a comment... they are nice for making notes!
echo "Our script worked!"
```

When we run our script, the output should be unchanged from before!

Shell variables

One important concept that we'll need to cover are shell variables. Variables are a great way of saving information under a name you can access later. In programming languages like Python and R, variables can store pretty much anything you can think of. In the shell, they usually just store text. The best way to understand how they work is to see them in action.

To set a variable, simply type in a name containing only letters, numbers, and underscores, followed by an `=` and whatever you want to put in the variable. Shell variable names are often uppercase by convention (but do not have to be).

```
$ VAR="This is our variable"
```

To use a variable, prefix its name with a `$` sign. Note that if we want to simply check what a variable is, we should use echo (or else the shell will try to run the contents of a variable).

```
$ echo $VAR
```

```
This is our variable
```

Let's try setting a variable in our script and then recalling its value as part of a command. We're going to make it so our script runs `wc -l` on whichever file we specify with `FILE`.

Our script:

```
#!/bin/bash

# set our variable to the name of our GTF file
FILE=dmel-all-r6.19.gtf

# call wc -l on our file
wc -l $FILE
```

```
$ bash demo.sh
```

```
542048 dmel-all-r6.19.gtf
```

What if we wanted to do our little `wc -l` script on other files without having to change `$FILE` every time we want to use it? There is actually a special shell variable we can use in scripts that allows us to use arguments in our scripts (arguments are extra information that we can pass to our script, like the `-l` in `wc -l`).

To use the first argument to a script, use `$1` (the second argument is `$2`, and so on). Let's change our script to run `wc -l` on `$1` instead of `$FILE`. Note that we can also pass all of the arguments using `$@` (not going to use it in this lesson, but it's something to be aware of).

Our script:

```
#!/bin/bash

# call wc -l on our first argument
wc -l $1
```

```
$ bash demo.sh dmel_unique_protein_isoforms_fb_2016_01.tsv
```

```
22129 dmel_unique_protein_isoforms_fb_2016_01.tsv
```

Nice!

Capturing the output of commands

One thing to be aware of when using variables: they are all treated as pure text. How do we save the output of an actual command like `ls` ?

A demonstration of what doesn't work:

```
$ TEST=ls
$ echo $TEST
```

```
ls
```

What does work (we need to surround any command with `$(command)`):

```
$ TEST=$(ls -l)
$ echo $TEST
```

```
total 90372 -rw-rw-r--. 1 jeff jeff 12534006 Jan 16 18:50 bash-lesson.tar.gz -rwxrwxr-x. 1 jeff jeff 40
Jan 1619:41 demo.sh -rw-rw-r--. 1 jeff jeff 77426528 Jan 16 18:50 dmel-all-r6.19.gtf -rw-r--r--. 1 jeff
jeff 721242 Jan 25 2016 dmel_unique_protein_isoforms_fb_2016_01.tsv drwxrwxr-x. 2 jeff jeff 4096 Jan 16
19:16 fastq -rw-r--r--. 1 jeff jeff 1830516 Jan 25 2016 gene_association.fb.gz -rw-rw-r--. 1 jeff jeff 1
5 Jan 16 19:17 test.txt -rw-rw-r--. 1 jeff jeff 245 Jan 16 19:24 word_counts.txt
```

Note that everything got printed on the same line. This is a feature, not a bug, as it allows us to use `$(commands)` inside lines of script without triggering line breaks (which would end our line of code and execute it prematurely).

Loops

To end our lesson on scripts, we are going to learn how to write a for-loop to execute a lot of commands at once. This will let us do the same string of commands on every file in a directory (or other stuff of that nature).

for-loops generally have the following syntax:

```
#!/bin/bash

for VAR in first second third
do
    echo $VAR
done
```

When a for-loop gets run, the loop will run once for everything following the word `in` . In each iteration, the variable `$VAR` is set to a particular value for that iteration. In this case it will be set to `first` during the first iteration, `second` on the second, and so on. During each iteration, the code between `do` and `done` is performed.

Let's run the script we just wrote (I saved mine as `loop.sh`).

```
$ bash loop.sh
```

```
first
second
third
```

What if we wanted to loop over a shell variable, such as every file in the current directory? Shell variables work perfectly in for-loops. In this example, we'll save the result of `ls` and loop over each file:

```
#!/bin/bash

FILES=$(ls)
for VAR in $FILES
do
        echo $VAR
done
```

```
$ bash loop.sh
```

```
bash-lesson.tar.gz
demo.sh
dmel_unique_protein_isoforms_fb_2016_01.tsv
dmel-all-r6.19.gtf
fastq
gene_association.fb.gz
loop.sh
test.txt
word_counts.txt
```

There's actually even a shortcut to run on all files of a particular type, say all .gz files:

```
#!/bin/bash

for VAR in *.gz
do
    echo $VAR
done
```

```
bash-lesson.tar.gz
gene_association.fb.gz
```

Writing our own scripts and loops

`cd` to our `fastq` directory from earlier and write a loop to print off the name and top 4 lines of every fastq file in that directory.

Is there a way to only run the loop on fastq files ending in `_1.fastq` ?

Solution

Create the following script in a file called `head_all.sh`

```
#!/bin/bash

for FILE in *.fastq
do
   echo $FILE
   head -n 4 $FILE
done
```

The "for" line could be modified to be `for FILE in *_1.fastq` to achieve the second aim

Concatenating variables

Concatenating (i.e. mashing together) variables is quite easy to do. Add whatever you want to concatenate to the beginning or end of the shell variable after enclosing it in `{}` characters.

```
FILE=stuff.txt
echo ${FILE}.example
```

```
stuff.txt.example
```

Can you write a script that prints off the name of every file in a directory with ".processed" added to it?

Solution

Create the following script in a file called `process.sh`

```
#!/bin/bash

for FILE in *
do
   echo ${FILE}.processed
done
```

Note that this will also print directories appended with ".processed". To truely only get files and not directories, we need to modify this to use the `find` command to give us only files in the current directory:

```
#!/bin/bash

for FILE in $(find . -max-depth 1 -type f)
do
   echo ${FILE}.processed
done
```

but this will have the side-effect of listing hidden files too.

Running a script like a linux command

We can save a little typing time by running our script like any other Linux command (eg ls) instead of having to type 'bash' before each command. This takes a bit of set up.

If we type:

```
$ demo.sh
```

```
bash: demo.sh: command not found...
```

Bash can't find our script. As it turns out, Bash will only look in certain directories for scripts to run. To run anything else, we need to tell Bash exactly where to look. To run a script that we wrote ourselves, we need to specify the full path to the file, followed by the filename. We could do this one of two ways: either with our absolute path `/lustre/home/tc008/yourUserName/demo.sh`, or with the relative path `./demo.sh`.

```
$ ./demo.sh
```

```
bash: ./demo.sh: Permission denied
```

There's one last thing we need to do. Before a file can be run, it needs "permission" to run. Let's look at our file's permissions with `ls -l`:

```
$ ls -l
```

```
-rw-rw-r--. 1 yourUsername tc001 12534006 Jan 16 18:50 bash-lesson.tar.gz
-rw-rw-r--. 1 yourUsername tc001       40 Jan 16 19:41 demo.sh
-rw-rw-r--. 1 yourUsername tc001 77426528 Jan 16 18:50 dmel-all-r6.19.gtf
-rw-r--r--. 1 yourUsername tc001   721242 Jan 25  2016 dmel_unique_protein_isoforms_fb_2016_01.tsv
drwxrwxr-x. 2 yourUsername tc001     4096 Jan 16 19:16 fastq
-rw-r--r--. 1 yourUsername tc001  1830516 Jan 25  2016 gene_association.fb.gz
-rw-rw-r--. 1 yourUsername tc001       15 Jan 16 19:17 test.txt
-rw-rw-r--. 1 yourUsername tc001      245 Jan 16 19:24 word_counts.txt
```

That's a huge amount of output. Let's see if we can understand what it is, working left to right.

- **1st column - Permissions:** On the very left side, there is a string of the characters `d`, `r`, `w`, `x`, and `-`. The `d` indicates if something is a directory (there is a `-` in that spot if it is not a directory). The other `r`, `w`, `x` bits indicates permission to **R**ead **W**rite and e**X**ecute a file. There are three columns of `rwx` permissions following the spot for `d`. If a user is missing a permission to do something, it's indicated by a `-`.
  - The first column of `rwx` are the permissions that the owner has (in this case the owner is `yourUsername`).
  - The second set of `rwx` s are permissions that other members of the owner's group share (in this case, the group is named `tc001`).
  - The third set of `rwx` s are permissions that anyone else with access to this computer can do with a file. Though files are typically created with read permissions for everyone, typically the permissions on your home directory prevent others from being able to access the file in the first place.
- **2nd column – Hard links:** The number of hard links, ie the number of files which point to the same location on disk as this file does.
- **3rd column - Owner:** This is the username of the user who owns the file. Their permissions are indicated in the first permissions column.
- **4th column - Group:** This is the user group of the user who owns the file. Members of this user group have permissions indicated in the second permissions column.
- **5th column - Size of file:** This is the size of a file in bytes, or the number of files/subdirectories if we are looking at a directory. (We can use the `-h` option here to get a human-readable file size in megabytes, gigabytes, etc.)
- **6th column - Time last modified:** This is the last time the file was modified.
- **7th column - Filename:** This is the filename.

So how do we change permissions? As I mentioned earlier, we need permission to execute our script. Changing permissions is done with `chmod`. To add executable permissions for all users we could use this:

```
$ chmod +x demo.sh
$ ls -l
```

```
-rw-rw-r--. 1 yourUsername tc001 12534006 Jan 16 18:50 bash-lesson.tar.gz
-rwxrwxr-x. 1 yourUsername tc001       40 Jan 16 19:41 demo.sh
-rw-rw-r--. 1 yourUsername tc001 77426528 Jan 16 18:50 dmel-all-r6.19.gtf
-rw-r--r--. 1 yourUsername tc001   721242 Jan 25  2016 dmel_unique_protein_isoforms_fb_2016_01.tsv
drwxrwxr-x. 2 yourUsername tc001     4096 Jan 16 19:16 fastq
-rw-r--r--. 1 yourUsername tc001  1830516 Jan 25  2016 gene_association.fb.gz
-rw-rw-r--. 1 yourUsername tc001       15 Jan 16 19:17 test.txt
-rw-rw-r--. 1 yourUsername tc001      245 Jan 16 19:24 word_counts.txt
```

Now that we have executable permissions for that file, we can run it.

```
$ ./demo.sh
```

```
Our script worked!
```

Special permissions

What if we want to give different sets of users different permissions. `chmod` actually accepts special numeric codes instead of stuff like `chmod +x`. The numeric codes are as follows: read = 4, write = 2, execute = 1. For each user we will assign permissions based on the sum of these permissions (must be between 7 and 0).

Let's make an example file and give everyone permission to do everything with it.

```
touch example
ls -l example
chmod 777 example
ls -l example
```

How might we give ourselves permission to do everything with a file, but allow no one else to do anything with it.

Solution

```
chmod 700 example
```

We want all permissions so: 4 (read) + 2 (write) + 1 (execute) = 7 for user (first position), no permissions, i.e. 0, for group (second position) and all (third position).

Key Points

- A shell script is just a list of bash commands in a text file.
- You can use variables and loops to automate tasks that were previously done by hand
- `chmod +x script.sh` will give a script permission to execute.

(../04-wildcards-pipes/index.html)

(../05
coffe

Licensed under CC-BY 4.0 (https://creativecommons.org/licenses/by/4.0/) 2018–2020 by The Carpentries (https://carpentries.org/)
Licensed under CC-BY 4.0 (https://creativecommons.org/licenses/by/4.0/) 2016–2018 by Software Carpentry Foundation (https://software-carpentry.org)

Edit on GitHub (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/edit/gh-pages/_episodes/05-scripts.md) / Contributing (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/blob/gh-pages/CONTRIBUTING.md) / Source (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/) / Cite (https://github.com/aniabrown/hpc-carpentry-shell-WHPC/blob/gh-pages/CITATION) / Contact (mailto:team@carpentries.org)

Using The Carpentries style (https://github.com/carpentries/styles/) version 9.5.3 (https://github.com/carpentries/styles/releases/tag/v9.5.3).