UNIX I: Logging in and finding your way around

This class uses the UNIX command line to do genomics and evolutionary analysis with bioinformatics tools. Today is about getting onto the UCR HPCC cluster and becoming comfortable moving around it.

What you'll learn today

- what the cluster is, and the difference between the login node and the compute nodes
- how to log in with `ssh` (password + Duo) or through the web with OnDemand (Jupyter, RStudio, VS Code)
- how the filesystem is organized: your home folder, your bigdata folder, and paths (absolute and relative, `~`, `.`, `..`)
- how to move around and look at files: `pwd`, `ls`, `cd`, `mkdir`, `cat`, `less`, `head`, `tail`, `wc`, `echo`
- how to get help (`man`, `--help`) and the keyboard shortcuts that save typing
- how to edit a file with `nano`, and write and run your first shell script

A quick reference table of every command in this lecture is at the end, followed by practice exercises.

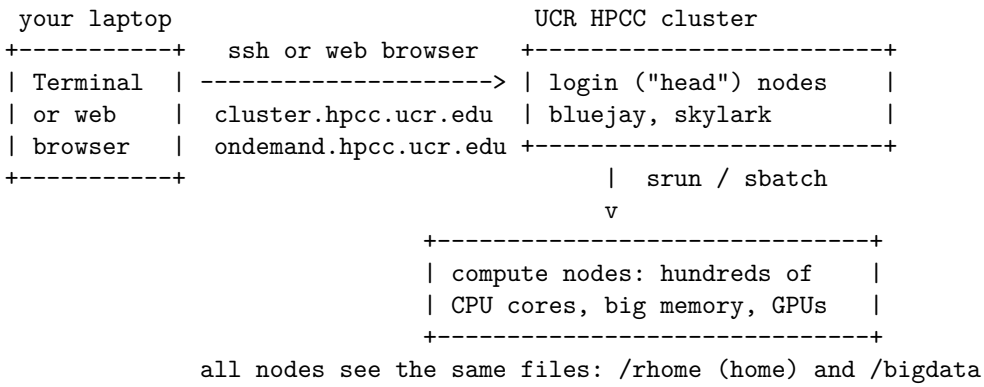
Where this fits. The other UNIX lectures build on this one:

Lecture	Topics
UNIX I (this one)	logging in, the filesystem, looking at files, nano, first script
UNIX II	copy/move/delete, wildcards, permissions, disk space, redirection and pipes, `grep`, compression, downloading and transferring files, modules, `srun`, Git
UNIX III + lab	variables, quoting, `if`, loops, scripts with arguments; lab: `sort`/`cut`/`uniq`/`sed`/`awk`
UNIX IV	SLURM batch jobs and job arrays, scratch storage, `tmux`, conda, containers, `find`/`xargs`

Guides you will use alongside these lectures: SSH keys guide and Git and GitHub guide.

The cluster: what you are connecting to

A **cluster** is many computers ("nodes") connected together and sharing the same storage. You never sit in front of them; you connect over the network from your laptop.



- When you `ssh` to `cluster.hpcc.ucr.edu` you land on one of the **login nodes** (also called head nodes). The address automatically sends you to whichever one is available, which is why the name in your prompt (e.g. `bluejay` or `skylark`) can change between logins.
- The login nodes are shared by *everyone* who is logged in. Use them for editing files, looking at data, moving files, and submitting jobs. HPCC asks that anything you run there stay small (under 100% of one CPU and under 1 GB of memory).

- Real work (assembling a genome, running BLAST on thousands of sequences, anything that runs for more than a few minutes) goes to the **compute nodes**, through the job scheduler, SLURM. You will start an interactive session on a compute node with `srun` in UNIX II and submit batch jobs with `sbatch` in UNIX IV. Heavy programs left running on a login node slow it down for everyone and may be killed by the administrators.
- Because every node mounts the same storage, a file you save on the login node is immediately visible to your jobs on the compute nodes. There is no need to copy files between the machines of the cluster.

The cluster runs Linux (Rocky Linux 8) and the command language you type is `bash`. The HPCC documentation is at https://hpcc.ucr.edu/manuals/; the cluster introduction lists the hardware.

Logging in

A terminal program

You need a terminal program on your own computer:

- **Mac**: the built-in **Terminal** (in Applications/Utilities), or iTerm2.
- **Windows**: MobaXterm (choose the free "Home Edition"; the portable version does not need administrator rights) is recommended by HPCC. Windows 10 and 11 also have `ssh` built into **PowerShell**. PuTTY is outdated and not recommended.
- **Linux or ChromeOS** (with Linux apps turned on): the default terminal.

Log in with ssh

`ssh` ("secure shell") opens an encrypted connection to another computer and gives you a command line on it. Replace `USERNAME` with your HPCC username (for UCR users, this is your NetID):

```
ssh USERNAME@cluster.hpcc.ucr.edu
```

1. The first time you connect, `ssh` asks whether you trust the computer (`Are you sure you want to continue connecting (yes/no/[fingerprint])?`). Type `yes`. This is only asked once per computer.
2. Type your **password** and press Enter. Nothing appears on the screen while you type a password, not even `*`. This is normal.
3. Follow the **Duo** two-factor instructions printed on the screen (for example, approve the push on your phone).

HPCC requires password + Duo, or an SSH key, for every login and file transfer. Password + Duo only works if your HPCC username is the same as your UCR NetID.

After you log in you will see a welcome message from HPCC and then a **prompt**, which will look similar to:

```
USERNAME@skylark:~$
```

The prompt shows your username, the name of the node you are on, and the folder you are in (`~` means your home folder; more below). The prompt ends with `$` and waits for you to type a command. In these notes, commands you type are shown in grey boxes *without* the prompt, so you can copy them.

Try a few commands to see who and where you are:

```
whoami      # your username
hostname    # the name of the computer you are on
pwd         # the folder you are in
date        # the current date and time
```

First login: change your password

New accounts get a temporary password by email. After your first login (or after a password reset) HPCC requires you to change it:

```
passwd
```

It asks for your current password once and the new one twice (again, nothing is shown as you type). The new password needs at least 8 characters and 3 of: lowercase, uppercase, number, punctuation.

Logging out

Type `exit`, or press **Ctrl-D**. Closing the terminal window also disconnects you, and anything still running in that window is stopped. (To keep work running after you disconnect, use a batch job or `tmux`; see UNIX IV.)

If you can't log in

- **Password not accepted / Duo not offered**: after too many wrong passwords, or if you did not change the initial password, the account is locked. Email support@hpcc.ucr.edu to ask for a password reset.
- **Your HPCC username is not your NetID**: Duo will not work. Ask support@hpcc.ucr.edu to change your username, or use SSH keys.
- **`Connection timed out`**: check your network connection and the spelling of `cluster.hpcc.ucr.edu`.

Graphics over ssh (optional)

Adding `-X` forwards graphical windows (X11) from the cluster to your screen, for the occasional program that opens a window (an image viewer, `gvim`, `emacs` in a window):

```
ssh -X USERNAME@cluster.hpcc.ucr.edu
```

MobaXterm and Linux handle this automatically. On a Mac you must install and run XQuartz first. You don't need `-X` for this class; the "HPCC Desktop" app in OnDemand (below) is an easier way to run graphical programs.

SSH keys: skip the password and Duo

An **SSH key** is a pair of files made on your laptop: a *private key* that never leaves your laptop, and a *public key* that you copy to the cluster (into `~/.ssh/authorized_keys`). Once set up, `ssh` logs you in without asking for your password or Duo. The same public key also lets you push code to GitHub, which you will need for the homework. You do this once per laptop.

Follow the SSH keys guide: its quick start is three commands on Mac/Linux or PowerShell/MobaXterm on Windows, and it also shows how to set up a shortcut so that you can type just `ssh hpcc`. Setting up keys is part of Homework 0.

Web access: OnDemand (Jupyter, RStudio, VS Code)

You can also use the cluster entirely from a web browser with **Open OnDemand**:

- Go to https://ondemand.hpcc.ucr.edu/ and log in with your cluster username and password, then Duo.
- **Files** menu: browse your folders, and upload or download files.
- **Shell access**: a terminal on the cluster in a browser tab, the same as logging in with `ssh`.
- **Interactive Apps** menu: start **Jupyter Notebook**, **RStudio Server**, **VS Code**, or an **HPCC Desktop** (a full graphical desktop). You pick the resources (CPUs, memory), the time limit and the partition, then click **Launch**. Your session waits in the queue, starts on a *compute node*, and a **Connect** button appears when it is ready.

Because OnDemand apps run as jobs on compute nodes, you get dedicated resources, but the session ends when its time limit is reached, and it holds those resources until then. Delete sessions you are finished with from the "My Interactive Sessions" page. (The older addresses `jupyter.hpcc.ucr.edu` and `rstudio.hpcc.ucr.edu` now redirect to the OnDemand instructions.)

Jupyter gives you a file browser, a text editor, a terminal (under **New** -> **Terminal**) and notebooks for Python or R. **RStudio** is the environment for the R part of the class. The terminal inside Jupyter, RStudio or VS Code is a regular `bash` shell on the cluster: every command in these lectures works there.

VS Code can run in the browser through OnDemand, or on your laptop connected to the cluster with a "Remote Tunnel" (see the HPCC VS Code page). HPCC asks that you **not** use the VS Code "Remote - SSH" extension, because it runs a server on the shared login node.

Notebooks are great for trying things out, but the code you turn in for homework will be plain text files (`.sh` shell scripts and `.py` Python scripts), not `.ipynb` notebook files. You can write those plain text files in the Jupyter, RStudio or VS Code editors, or with `nano` (below).

If you want to try R or Python without the cluster, Posit Cloud (formerly rstudio.cloud) and Try Jupyter run in a browser. Keep your code on GitHub so it is easy to move between these places.

The command line

The program that reads what you type at the prompt and runs it is called the **shell**; on the cluster it is `bash`. Every command has the same shape:

```
command  -options  arguments
   ls       -l     /bigdata/gen220/shared
```

- the **command** is the program to run (`ls` lists files),
- **options** (also called flags) change how it behaves; they start with `-` (short, single letter: `-l`) or `--` (long: `--help`). Short options can be combined: `ls -l -h` is the same as `ls -lh`,
- **arguments** are what the command works on, usually file or folder names.

Separate each part with spaces. Everything is **case sensitive**: `ls` is a command, `LS` is not, and `Data.txt` and `data.txt` are different files. Anything after a `#` is a **comment** and is ignored, so the notes below use `#` to explain commands.

Files, folders and paths

The directory tree

Files are organized in **directories** (folders), which can contain other directories. The whole system is one upside-down tree starting at the **root**, written `/`. The parts you will use on the cluster:

```
/                                  the root: top of the tree
|-- rhome/
|   `-- USERNAME/                  your HOME directory (~), 50 GB limit
|       |-- bigdata  -> /bigdata/gen220/USERNAME     (a shortcut)
|       `-- shared   -> /bigdata/gen220/shared       (a shortcut)
|-- bigdata/
|   `-- gen220/                    the "lab" everyone in this class belongs to
|       |-- USERNAME/              your bigdata folder: put data and results here
|       `-- shared/                files shared with the whole class
|           `-- simple/            small example files used in these lectures
|-- tmp/                           temporary files, local to each node
`-- opt/, usr/, bin/ ...           the operating system and installed software
```

Location	What it is for	Space
`/rhome/USERNAME` (`~`)	scripts, settings, small files; where you start each login	50 GB per user
`/bigdata/gen220/USERNAME` (`~/bigdata`)	data, large results, homework work area	shared by the lab (quota bought by the lab)
`/bigdata/gen220/shared` (`~/shared`)	files the instructor shares with the class	lab quota

If you were already in a research lab on HPCC, your bigdata folder is `/bigdata/YOURLAB/USERNAME` instead, and `~/bigdata` points there. How to check how much space you are using (`check_quota`, `du`) is covered in UNIX II; fast temporary `/scratch` space is covered in UNIX IV. See also the HPCC data storage page.

Paths: absolute and relative

A **path** is the address of a file or folder.

- An **absolute path** starts with `/` and spells out every folder from the root: `/bigdata/gen220/shared/simple/rice_random` It means the same thing no matter where you are.
- A **relative path** does not start with `/` and is read starting from your **current working directory** (the folder you are "in"). If you are in `/bigdata/gen220/shared`, then `simple/rice_random_exons.bed` refers to the same file as the absolute path above.

A few special names are shortcuts in paths:

Name	Means	Example
`/`	the root of the tree (at the start of a path); separates folder names elsewhere	`/bigdata/gen220`
`~`	your home directory, `/rhome/USERNAME`	`~/bigdata`
`.`	the current directory	`./myscript.sh`
`..`	the directory one level up (the "parent")	`../..` is two levels up
`-`	(only with `cd`) the directory you were in before	`cd -`

Tips for naming files and folders: avoid spaces (use `_` or `-`: `rice_exons.bed`, not `rice exons.bed`), and avoid characters such as `* ? ! & ( ) ' "`. They all have special meanings to the shell and make files annoying to work with. Files whose names start with a `.` (like `.bashrc` or `.ssh`) are **hidden**: `ls` doesn't show them unless you ask with `ls -a`.

Moving around

pwd: where am I?

`pwd` (print working directory) prints the absolute path of the folder you are in. When you log in you start in your home directory:

```
pwd
```

```
/rhome/USERNAME
```

ls: what is here?

`ls` lists the contents of a folder: the current one, or the folders you name.

```
ls                                  # the current folder
ls /bigdata/gen220/shared/simple    # another folder (absolute path)
ls -l /bigdata/gen220/shared/simple # long listing, with details
```

The long listing will look similar to:

```
lrwxrwxrwx 1 jstajich gen220    20 Sep 30 12:06 gene_names.txt -> yeast_gene_names.txt
-rw-r--r-- 1 jstajich gen220   603 Oct 10  2018 numbers_floating.dat
-rw-r--r-- 1 jstajich gen220 22447 Oct 10  2018 rice_random_exons.bed
-rw-rw-r-- 1 jstajich gen220 33894 Sep 30 12:05 yeast_gene_names.txt
```

Reading one line of `ls -l` from left to right:

```
-rw-r--r--  1  jstajich  gen220  22447  Oct 10  2018  rice_random_exons.bed
|           |  |         |       |      |             |
|           |  owner     group   size   last modified name
|           number of links      (bytes)
type and permissions: first letter - is a file, d a directory, l a link
```

The `l` entry, `gene_names.txt -> yeast_gene_names.txt`, is a **symbolic link**: a shortcut that points to another file. (Permissions and links are explained in UNIX II.)

Useful options (combine them freely, e.g. `ls -lhtr`):

Option	What it does
`-l`	long listing: permissions, owner, size, date
`-h`	with `-l`, human-readable sizes (`22K` instead of `22447`)
`-a`	show all files, including hidden ones starting with `.`
`-t`	sort by time last modified, newest first
`-r`	reverse the order (`-ltr`: newest at the bottom, handy in a big folder)
`-S`	sort by size, largest first
`-F`	add a `/` after directory names and `@` after links
`-R`	also list everything inside subfolders (recursive)
`-d`	list a directory itself, not its contents (`ls -ld ~`)

```
ls -lh /bigdata/gen220/shared/simple
```

```
lrwxrwxrwx 1 jstajich gen220  20 Sep 30 12:06 gene_names.txt -> yeast_gene_names.txt
-rw-r--r-- 1 jstajich gen220 603 Oct 10  2018 numbers_floating.dat
-rw-r--r-- 1 jstajich gen220 22K Oct 10  2018 rice_random_exons.bed
-rw-rw-r-- 1 jstajich gen220 34K Sep 30 12:05 yeast_gene_names.txt
```

cd: change directory

`cd` moves you to another folder. It prints nothing if it works; use `pwd` or look at your prompt to see where you are.

Command	Goes to
`cd /bigdata/gen220/shared`	an absolute path
`cd simple`	a folder called `simple` inside the current folder (relative path)
`cd ..`	up one level, to the parent folder
`cd ../..`	up two levels
`cd` or `cd ~`	your home directory, from anywhere
`cd ~/bigdata`	your bigdata folder
`cd -`	back to the folder you were in before (and prints its name)

Follow along:

```
cd /bigdata/gen220/shared
pwd
cd simple
pwd
cd ..
```

```
pwd
cd ../..
pwd
cd -
cd
pwd
```

```
/bigdata/gen220/shared
/bigdata/gen220/shared/simple
/bigdata/gen220/shared
/bigdata
/bigdata/gen220/shared
/rhome/USERNAME
```

If you get `No such file or directory`, check the spelling and capitalization, and use `pwd` and `ls` to check where you are: a relative path only works from the right place.

mkdir and rmdir: make and remove folders

```
cd ~/bigdata
mkdir unix1                    # make one folder
mkdir Alpha/Beta/Zeta          # fails: Alpha and Alpha/Beta don't exist yet
mkdir -p Alpha/Beta/Zeta       # -p makes all the missing parent folders
ls -R Alpha
```

```
mkdir: cannot create directory 'Alpha/Beta/Zeta': No such file or directory
Alpha:
Beta

Alpha/Beta:
Zeta

Alpha/Beta/Zeta:
```

`mkdir -p` also doesn't complain if the folder already exists, which makes it safe to use in scripts.

`rmdir` removes a folder, but only if it is empty:

```
rmdir Alpha/Beta/Zeta Alpha/Beta Alpha
```

Deleting files and folders that are not empty (`rm`), and copying and moving (`cp`, `mv`), are covered in UNIX II.

pushd and popd: bookmarks for folders

`pushd` is like `cd`, but it remembers where you came from on a list (a "stack"). `popd` takes you back to the last folder you `pushd`-ed from. Both print the stack, with the current folder first:

```
pushd /bigdata/gen220      # go here, remember where we were
pushd /tmp                 # go here, remember /bigdata/gen220 too
cd                         # cd moves around without changing the stack
popd                       # back to /bigdata/gen220
popd                       # back to where we started
```

```
/bigdata/gen220 ~
/tmp /bigdata/gen220 ~
/bigdata/gen220 ~
~
```

This differs from `cd -`, which only remembers the one previous folder. Type `dirs` to see the stack.

realpath: the full path to a file

`realpath` prints the absolute path of a file, and follows symbolic links to the original file:

```
cd /bigdata/gen220/shared/simple
realpath rice_random_exons.bed
realpath gene_names.txt
```

```
/bigdata/gen220/shared/simple/rice_random_exons.bed
/bigdata/gen220/shared/simple/yeast_gene_names.txt
```

This is handy when you need to give a program (or a homework answer) the full path to a file.

Looking at files

Most bioinformatics data (FASTA, FASTQ, BED, GFF, VCF, CSV and tab-separated tables) are plain **text** files, so the tools below work on all of them. The examples use `rice_random_exons.bed`, a BED file listing 1000 exons on the rice genome: each line has a chromosome name, a start position and an end position, separated by tabs.

file: what kind of file is this?

```
cd /bigdata/gen220/shared/simple
file rice_random_exons.bed
```

```
rice_random_exons.bed: ASCII text
```

`file` will also tell you if something is a compressed (`gzip compressed data`) or a program (`ELF 64-bit ...`), which you should not try to print to the screen.

cat: print the whole file

`cat` prints the whole file to the screen. That is fine for a short file; for a long one it scrolls past faster than you can read (press **Ctrl-C** to stop it). Use `less` or `head` instead.

```
cat ~/.bashrc
```

(`cat` stands for concatenate: `cat file1 file2` prints one file after the other. You will use it to combine files in UNIX II.)

less and more: page through a file

`less` shows a file one screen at a time. It is the best way to look at a large file, because it doesn't load the whole file first.

```
less /bigdata/gen220/shared/simple/yeast_gene_names.txt
```

Keys to use inside `less`:

Key	Action
Space or `f` / `b`	forward / back one page
Down / Up arrow	one line at a time
`g` / `G`	jump to the start / end of the file
`/text` then Enter	search forward for `text` (all matches are highlighted)
`n` / `N`	next / previous match
`-S`	turn line wrapping on or off (useful for wide tables)
`h`	help
`q`	quit

`more` is the older version of `less`: Space for the next page, `/` to search, `q` to quit, but you can't go backwards. (Hence the joke: "less is more".) The same keys work when you read manual pages with `man`, which uses `less` to display them.

head and tail: the beginning or end of a file

`head` prints the first 10 lines of a file; `-n` picks how many. It is the quickest way to see what is in a file, or what the column headers of a table are.

```
head -n 3 /bigdata/gen220/shared/simple/rice_random_exons.bed
```

```
Chr7    21408673    21408826
Chr9    16031526    16031938
Chr11   4762531 4762595
```

`tail` prints the last 10 lines (or `-n` lines). It is useful for the last messages in a log file, e.g. to see whether a program finished or crashed.

```
tail -n 3 /bigdata/gen220/shared/simple/rice_random_exons.bed
```

```
Chr2    16491612    16491671
Chr2    32686077    32686828
Chr2    19870914    19870999
```

Two more `tail` tricks:

- `tail -n +998 FILE` prints from line 998 to the end (a `+` means "starting at line").
- `tail -f logfile.txt` keeps printing new lines as they are added to the file, so you can watch a running program's log. Press **Ctrl-C** to stop watching.

Given more than one file (`head -n 2 file1 file2`), `head` and `tail` print each one under a `==> file1 <==` header line.

wc: count lines, words and characters

`wc` (word count) prints the number of lines, words and bytes (characters) in a file. `wc -l` counts only the lines, which is one of the commands you will use most: one line per exon, per gene, per BLAST hit, per SNP...

```
wc rice_random_exons.bed
wc -l rice_random_exons.bed
```

```
 1000  3000 22447 rice_random_exons.bed
1000 rice_random_exons.bed
```

So there are 1000 exons (lines), 3000 "words" (3 columns x 1000 lines) and 22447 bytes (the same size `ls -l` reported).

echo: print a message

`echo` prints whatever you give it. It is how scripts report what they are doing, and a quick way to see the value of a **variable** (a name that stores a value; variables start with `$`):

```
echo "hello there"
echo "My home directory is $HOME and my username is $USER"
echo -e "Chrom\tStart\tEnd"      # -e turns \t into a tab and \n into a new line
```

```
hello there
My home directory is /rhome/USERNAME and my username is USERNAME
Chrom   Start   End
```

Without `-e`, `echo "Chrom\tStart"` prints the `\t` literally. You will write your own variables in UNIX III.

Getting help

Nobody remembers every option. These are the ways to look them up:

```
man ls          # the manual page for ls (uses less: Space, /search, q to quit)
ls --help       # a shorter summary of the options (most programs have this)
help cd         # help for commands built into bash itself (cd, pushd, echo, ...)
type cd         # is this a program, or built into the shell?
```

```
cd is a shell builtin
```

A manual page starts with a **SYNOPSIS** showing how the command is used (`ls [OPTION]... [FILE]...`: square brackets mean "optional", `...` means "you can give more than one"), then a **DESCRIPTION** listing every option. Search it with `/`, e.g. type `/-h` in `man ls` to find the `-h` option.

Bioinformatics programs usually print their help with `-h`, `--help`, or when run with no arguments. On the web, https://explainshell.com breaks a command line into its pieces and explains each one, and https://tldr.sh has short, example-based summaries of common commands. The HPCC Linux Basics manual is another reference.

Keyboard shortcuts

These save a lot of typing, and fixing typos.

Keys	Action
Tab	complete a command, file or folder name. Press Tab twice to list all the matches
Up / Down arrow	step back / forward through commands you typed before
Ctrl-R, then type	search your command history; Ctrl-R again for an older match, Enter to run, Ctrl-G to give up
`history`	list your recent commands, numbered
`!!`	run the previous command again
`!123`	run command number 123 from `history`
Ctrl-C	stop (cancel) the command that is running, or throw away the line you are typing
Ctrl-D	end of input; at an empty prompt, log out
Ctrl-A / Ctrl-E	jump to the start / end of the line
Ctrl-W	delete the word before the cursor
Ctrl-U / Ctrl-K	delete from the cursor to the start / end of the line
Ctrl-L (or `clear`)	clear the screen

Use **Tab completion** all the time: type `ls /big`, press Tab and bash fills in `/bigdata/`; type `ls /bigdata/gen220/sh` and Tab gives `shared/`. If nothing happens, there is either no match or more than one: press Tab again to see the choices. It is faster than typing and it prevents spelling mistakes in long file names.

Copy and paste depends on your terminal: on a Mac Terminal use Cmd-C / Cmd-V; in MobaXterm, selecting text with the mouse copies it and Shift-Insert (or right-click) pastes; in a Linux terminal use Ctrl-Shift-C / Ctrl-Shift-V. Ctrl-C in a terminal does *not* copy: it cancels.

Editing text files with nano

You will write scripts and notes as plain text files. `nano` is the simplest editor on the command line. Open a file (it is created if it doesn't exist):

```
nano notes.txt
```

Type as in any editor, using the arrow keys to move. The commands are listed at the bottom of the screen; `^` means the Ctrl key, so `^X` is Ctrl-X.

Keys	Action
Ctrl-O, then Enter	save ("write Out")
Ctrl-X	exit (asks whether to save if there are changes; answer `Y` or `N`)
Ctrl-W	search ("Where is")
Ctrl-K / Ctrl-U	cut the current line / paste it
Ctrl-G	help

Other editors:

- `vim` (or `vi`) and `emacs` are powerful editors that many programmers use; both take some learning. If you end up in `vim` by accident, press Esc, type `:q!` and Enter to quit without saving. See the HPCC text editors page.
- The editors in **Jupyter**, **RStudio** and **VS Code** (through OnDemand, above) edit the files on the cluster directly, with a mouse and syntax highlighting.
- If you write a script on your own computer, use a code editor (VS Code, BBEdit, Notepad++), not a word processor like Word, and save it as plain text with UNIX (LF) line endings. Moving files to and from the cluster is covered in UNIX II.

Writing and running a shell script

A **shell script** is a text file containing the commands you would type at the prompt. Putting your commands in a script means you can re-run the whole analysis with one command, fix a mistake and run it again, and share exactly what you did. All the homework in this class will be turned in as scripts.

Make a small data file to practice on

```
mkdir -p ~/bigdata/scripts_practice
cd ~/bigdata/scripts_practice
echo -e "YAL001C\tTFC3" > genes.txt      # > creates (or overwrites) the file
echo -e "YAL002W\tVPS8" >> genes.txt     # >> adds to the end of the file
echo -e "YAL003W\tEFB1" >> genes.txt
cat genes.txt
```

```
YAL001C TFC3
YAL002W VPS8
YAL003W EFB1
```

`>` and `>>` send a command's output into a file instead of to the screen ("redirection"; more in UNIX II). Be careful: `>` replaces whatever was in the file.

Write the script

Open a new file in a text editor (`nano`, or the Jupyter, RStudio or VS Code editors):

```
nano count_lines.sh
```

Type in these lines. Save with **Ctrl-O** (then Enter) and exit with **Ctrl-X**.

```
#!/usr/bin/env bash
# count_lines.sh - report how many lines are in a file
echo "Counting the lines in genes.txt"
wc -l genes.txt
```

- The first line, starting with `#!` (called the "shebang"), tells the computer which program should run this file - here, `bash`.
- Every other line starting with `#` is a **comment**. It is ignored when the script runs; it's a note for people reading it (including you in a month).
- By convention, shell scripts end in `.sh`.

Run the script

There are two ways. The first is to pass the file to `bash`:

```
bash count_lines.sh
```

```
Counting the lines in genes.txt
3 genes.txt
```

The second way is to make the script **executable** (give it the `x` permission) with `chmod`, then run it directly:

```
chmod +x count_lines.sh
ls -l count_lines.sh      # the x's in -rwxr-xr-x mean it can be executed
./count_lines.sh
```

```
-rwxr-xr-x 1 USERNAME gen220 130 Sep 24 10:15 count_lines.sh
Counting the lines in genes.txt
3 genes.txt
```

Why `./`? When you type a command name like `ls`, the shell looks for a program with that name in a list of folders called your `PATH` (see it with `echo $PATH`). Your current folder is not on that list, so you have to say where the script is: `./` means "in this folder". Without it you get `command not found`. (File permissions are explained in UNIX II.)

Variables and arguments

A script that only works on `genes.txt` isn't very useful. Scripts can use **variables** and take **arguments** from the command line, so the same script works on any file:

```
#!/usr/bin/env bash
# file_summary.sh - print the number of lines and the first lines of a file
# usage: ./file_summary.sh FILENAME

if [ $# -lt 1 ]; then
    echo "usage: $0 FILENAME"
    exit 1
fi

FILE=$1
LINES=$(wc -l < "$FILE")
echo "$FILE has $LINES lines"
echo "The first 2 lines are:"
head -n 2 "$FILE"
```

```
chmod +x file_summary.sh
./file_summary.sh genes.txt
./file_summary.sh /bigdata/gen220/shared/simple/rice_random_exons.bed
./file_summary.sh
```

```
genes.txt has 3 lines
The first 2 lines are:
YAL001C TFC3
YAL002W VPS8
/bigdata/gen220/shared/simple/rice_random_exons.bed has 1000 lines
```

```
The first 2 lines are:
Chr7    21408673    21408826
Chr9    16031526    16031938
usage: ./file_summary.sh FILENAME
```

- `FILE=$1` creates a variable called `FILE`. There must be **no spaces** around the `=`: `FILE = $1` is an error, because bash thinks you are running a command called `FILE`.
- `$1` is the first argument given after the script name, `$2` the second, and so on. `$#` is the number of arguments, and `$0` is the name of the script itself.
- Use `$FILE` (with a `$`) to get the value back out. Put it in double quotes, `"$FILE"`, so file names containing spaces still work.
- `$(command)` runs a command and captures its output - here, the line count from `wc -l`. (`wc -l < file` prints just the number, without the file name.)
- The `if [ ... ]; then ... fi` block checks that an argument was given, and prints a usage message and stops (`exit 1`) if not.

Variables, quoting, `if` statements and loops are covered in detail in UNIX III.

Common problems

- **`Permission denied`** when running `./myscript.sh` - you forgot `chmod +x myscript.sh` (or use `bash myscript.sh`).
- **`command not found`** - you left off the `./`, or you have spaces around an `=`.
- **`bad interpreter`** or an error mentioning `\r` or `^M` - the file was saved with Windows line endings (e.g. edited in Notepad on Windows). Fix it on the cluster with `dos2unix myscript.sh`, and set your editor to use UNIX (LF) line endings.
- The script ran but did the wrong thing - run it with `bash -x myscript.sh` to print each command (marked with `+`) as it runs.

Practice: write a script called `count_genes.sh` which takes a file name as an argument and prints how many lines in the file contain the text `YAL` (hint: `grep -c YAL FILENAME`; `grep` is covered in UNIX II). Run it on `genes.txt`; it should print `3`.

Practice exercises

These use only the commands from today. Do them on the cluster (an `ssh` login, or a terminal in OnDemand).

1. Log in. Run `whoami`, `hostname` and `pwd`. Which login node are you on, and what is the absolute path of your home directory?
2. Use `ls -l ~` to look at your home directory. Where do the `bigdata` and `shared` links point? Check with `realpath ~/bigdata`.
3. Make a folder `~/bigdata/unix1` and `cd` into it. Use `pwd` to check. Then go back to your home directory with a single command, and return with `cd -`.
4. Without changing directory, list the files in `/bigdata/gen220/shared/simple`, with human-readable sizes. Which entry is a symbolic link, and to which file?
5. `cd /bigdata/gen220/shared/simple`. Now, where does `cd ../..` take you? Predict first, then check with `pwd`. Come back using a relative path.
6. How many exons (lines) are in `rice_random_exons.bed`?
7. What are the first 5 and the last 2 exons in the file?
8. Open `rice_random_exons.bed` in `less` and search for `Chr12`. What is the first exon on chromosome 12? Which line number is it on? (Hint: `less -N` shows line numbers.)
9. Use `man head` or `head --help` to find the option that prints the first *bytes* instead of lines. Print the first 20 bytes of the file.
10. Use the Up arrow or Ctrl-R to re-run your `wc -l` command from question 6 without retyping it. Then type `history` to see everything you have done.
11. In `~/bigdata/unix1`, use `nano` to write a script `exons_summary.sh` that prints the message `Rice exon file`, then the number of lines, then the first 3 lines of `/bigdata/gen220/shared/simple/rice_random_exons.bed`.

Run it with `bash`, then make it executable and run it with `./`.

12. Modify `file_summary.sh` (above) so that it also prints the last 2 lines of the file. Test it on `rice_random_exons.bed` and on `genes.txt`.

Preview of UNIX III: how many *different* chromosome names are in the first column of `rice_random_exons.bed`? You can't answer this easily with today's commands; in the UNIX III lab you will learn to answer it with one line: `cut -f 1 rice_random_exons.bed | sort | uniq | wc -l`.

Not on the cluster? `rice_random_exons.bed` is also in the `data/` folder of the class repository https://github.com/biodataprog/GEN220/tree/master/data, and more example data for the homework is in https://github.com/biodataprog/GEN220_data. (Downloading files and `git clone` are covered in UNIX II.)

Answers to check yourself: (6) 1000. (7) the first line is `Chr7 21408673 21408826`, the last is `Chr2 19870914 19870999`. (8) `Chr12 86793 86842`, on line 13. (9) `head -c 20`. (Preview) 13: `Chr1` to `Chr12`, plus one exon on `ChrSy`.

Quick reference

Command	What it does	Example
`ssh`	log in to another computer	`ssh USERNAME@cluster.hpcc.ucr.edu`
`exit`, Ctrl-D	log out	`exit`
`passwd`	change your password	`passwd`
`whoami`, `hostname`	your username; the computer you are on	`hostname`
`pwd`	print the current directory	`pwd`
`ls`	list files (`-l` long, `-h` sizes, `-a` hidden, `-t` by time, `-r` reverse)	`ls -lh ~/bigdata`
`cd`	change directory (`..` up, `~` home, `-` previous)	`cd /bigdata/gen220/shared`
`mkdir`	make a directory (`-p` with parents)	`mkdir -p proj/data`
`rmdir`	remove an empty directory	`rmdir proj/data`
`pushd`, `popd`, `dirs`	change directory and remember / go back / show the list	`pushd /tmp`
`realpath`	full absolute path of a file	`realpath gene_names.txt`
`file`	what kind of file	`file rice_random_exons.bed`
`cat`	print a whole file	`cat genes.txt`
`less`, `more`	page through a file (`/` search, `q` quit)	`less big_file.txt`
`head`, `tail`	first / last lines (`-n N`; `tail -f` follow)	`head -n 5 file.bed`
`wc`	count lines, words, bytes (`-l` lines only)	`wc -l file.bed`
`echo`	print a message or variable (`-e` for `\t`, `\n`)	`echo "$HOME"`
`man`, `--help`, `help`	manual page; option summary; help for bash builtins	`man ls`
`history`	list previous commands	`history`
`nano`	simple text editor (Ctrl-O save, Ctrl-X exit)	`nano script.sh`
`bash script.sh`	run a script	`bash count_lines.sh`
`chmod +x`	make a script executable, then run with `./`	`chmod +x count_lines.sh`

Learn more

There are many free tutorials for the UNIX command line in biology:

- Data Carpentry and Software Carpentry (part of The Carpentries), especially The Unix Shell
- Data Intensive Biology training, e.g. Shell Genomics
- Getting Started with Genomics Tools
- Happy Belly Bioinformatics
- HPCC manuals: Login, Linux Basics, Command line basics, File systems, Open OnDemand

Next: UNIX II - copying and moving files, pipes, running programs on the cluster, and Git.