
2014-argonne-soils-automation

Release 0.8

October 05, 2014

1	Automation Workshop Welcome	1
1.1	Schedule	1
1.2	Additional Resources	2
2	Introducing the Shell	3
2.1	Objectives	3
2.2	A Parable	3
2.3	Key Points	4
3	Files and Directories	5
3.1	Objectives	5
3.2	Cheat Sheet	5
3.3	Navigating the filesystem	5
3.4	Before you begin	5
3.5	Nelle's files	6
3.6	Key Points	10
3.7	Exercises	10
4	Creating Things	13
4.1	Objectives	13
4.2	Making Directories	13
4.3	Let's make a file	14
4.4	Which Editor?	14
4.5	Deleting Is Forever	15
4.6	With Great Power Comes Great Responsibility	15
4.7	More Practice	16
4.8	Another Useful Abbreviation	17
4.9	Key Points	17
4.10	Exercises	17
5	Pipes and Filters	19
5.1	Objectives	19
5.2	Multi-step operations with bash	19
5.3	Wildcards	20
5.4	Downlading data with redirection	22
5.5	Key Points	23
5.6	Exercises	24
6	Loops	27

6.1	Objectives	27
6.2	Measure Twice, Run Once	29
6.3	Key Points	31
6.4	Exercises	32
7	Shell Scripts	33
7.1	Objectives	33
7.2	Now we're cooking	33
7.3	Text vs. Whatever	34
7.4	Why Isn't It Doing Anything?	35
7.5	Key Points	37
7.6	Exercises	37
8	Getting started with Amazon EC2	39
8.1	Logging into your new instance "in the cloud"	39
8.2	Amazon Web Services reference material	40
9	Basic EC2, command line, and BLAST	41
9.1	Install BLAST and some other software	41
9.2	A few post-tutorial links	43
10	Mapping with bwa	45
10.1	Getting the Dependencies	45
10.2	Getting the Data	45
10.3	Mapping the Reads	46
10.4	Visualizing your Data with Tablet	47

Automation Workshop Welcome

Welcome to the automation workshop at the 6th Argonne Soils Meeting.

1.1 Schedule

Day	Schedule
Friday 10/03	<ul style="list-style-type: none"> • 1:30pm-5:00pm: Automating things with the shell (Will Trimble) <ul style="list-style-type: none"> – <i>Introducing the Shell</i> (What is going on?) – <i>Files and Directories</i> (cd,ls) – <i>Creating Things</i> (mv,rm,cp) – <i>Pipes and Filters</i> – <i>Loops</i> (for) – <i>Shell Scripts</i> (Doing Stuff) – novice-shell/06-find (find,grep)
Saturday 10/04	<ul style="list-style-type: none"> • 9:00 Introduction (Adina Howe) • 9:30 MEGAN (Daniel Huson) • 11:30 MG-RAST website(Folker Meyer) • 12:00 MG-RAST API(Daniel Braithwaite) <ul style="list-style-type: none"> – install-matR.txt • 12:30-1:30 Lunch • 1:30 KBase – metabolic models for metagenomes (Chris Henry) • 2:30-5:00pm Cloud computing to do more (Will Trimble) <ul style="list-style-type: none"> – Notes on cloud computing – Getting started with EC2: <i>Logging into your new instance “in the cloud”</i> – Lecture notes on BLAST – Similarity search: <i>Basic EC2, command line, and BLAST</i> • (Lengthy lecture notes on short read mapping) • Short read mapping: <i>Mapping with bwa</i>

1.2 Additional Resources

These bash lessons were adapted from Software Carpentry's bash for beginners lessons. Software Carpentry is a nonprofit that maintains [computational training materials for scientists](#).

The cloud computing lessons were adapted from GED's [ANGUS course](#) which is run at MSU every summer and teaches workshops like this. Specifically, this is a one-afternoon abbreviation of the [2013 CEMI workshop at Caltech](#), which was a four-day course.

Introducing the Shell

2.1 Objectives

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

2.2 A Parable

Nelle Nemo, a marine biologist, has just returned from a six-month survey of the [North Pacific Gyre](#), where she has been sampling gelatinous marine life in the [Great Pacific Garbage Patch](#). She has 300 samples in all, and now needs to:

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

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

The bad news is that if she has to run `goostat` and `goodiff` by hand, she'll have to enter filenames and click “OK” 45,150 times (300 runs of `goostat`, plus $300 \times 299/2$ runs of `goodiff`). At 30 seconds each, that will take more than two weeks. Not only would she miss her paper deadline, the chances of her typing all of those commands right are practically zero.

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

2.2.1 What and Why

At a high level, computers do four things:

- run programs;
- store data;
- communicate with each other; and
- interact with us.

They can do the last of these in many different ways; most of us use windows, icons, mice, and pointers. These technologies didn't become widespread until the 1980s, and, critically, *it is exceptionally difficult to automate mouseclicks*.

There exists an automatable, text-only, interactive interface under the surface of our computers and phones. Called a command-line interface, or CLI, to distinguish it from the graphical user interface, or GUI, that most people now use. The heart of a CLI is a read-evaluate-print loop, or REPL: when the user types a command and then presses the enter (or return) key, the computer reads it, executes it, and prints its output. The user then types another command, and so on until the user logs off.

This description makes it sound as though the user sends commands directly to the computer, and the computer sends output directly to the user. In fact, there is usually a program in between called a command shell. What the user types goes into the shell; it figures out what commands to run and orders the computer to execute them.

A shell is a program like any other. What's special about it is that its job is to run other programs rather than to do calculations itself. The most popular Unix shell is bash. Bash is the default shell on most modern implementations of Unix, and in most packages that provide Unix-like tools for Windows.

Using Bash or any other shell sometimes feels more like programming than like using a mouse. Commands are terse (often only a couple of characters long), their names are frequently cryptic, and their output is lines of text rather than something visual like a graph. On the other hand, the shell allows us to combine existing tools in powerful ways with only a few keystrokes and to set up pipelines to handle large volumes of data automatically. In addition, the command line is often the easiest way to interact with remote machines. As clusters and cloud computing become more popular for scientific data crunching, being able to drive them is becoming a necessary skill.

This no-frills text-only dialog with the computer can be your data file handling robot, downloading hundreds or thousands of data files and executing thousands or hundreds of thousands of programs on your behalf.

You just need some magic words.

2.3 Key Points

- A shell is a program whose primary purpose is to read commands and run other programs.
- The shell's main advantages are its high action-to-keystroke ratio, its support for automating repetitive tasks, and that it can be used to access networked machines.
- The shell's main disadvantages are its primarily textual nature and how cryptic its commands and operation can be.

Files and Directories

3.1 Objectives

- Explain the similarities and differences between a file and a directory.
- Translate an absolute path into a relative path and vice versa.
- Construct absolute and relative paths that identify specific files and directories.
- Explain the steps in the shell's read-run-print cycle.
- Identify the actual command, flags, and filenames in a command-line call.
- Demonstrate the use of tab completion, and explain its advantages.

3.2 Cheat Sheet

Quick guide to shell.

3.3 Navigating the filesystem

The part of the operating system responsible for managing files and directories is called the file system. It organizes our data into files, which hold information, and directories (also called “folders”), which hold files or other directories.

Several commands are frequently used to create, inspect, rename, and delete files and directories. To start exploring them, let's open a shell window:

\$

The dollar sign is a prompt, which shows us that the shell is waiting for input; your shell may show something more elaborate.

3.4 Before you begin

3.4.1 Windows

You need to install `git bash`.

After installing git bash, run the [Software Carpentry installer](#), which installs tools inside of gitbash that make it more usable.

3.4.2 Mac OS

You can *probably* get away without installing anything for the class; the unix-like tools you need are built in.

Just in case something important does not work, you may need the (free) apple Xcode Developer bundle. This should be available for free from the apple store or from [Apple's developer site](#).

3.5 Nelle's files

In these lessons, we're going to explore Nelle's files. You can download the example files and directories which Nelle is using, so that you can explore the same files as described in the lesson. To do this, download the zipped filesystem:

Download [filesystem.zip](#).

Unpack the zipped files - on Windows or a Mac, you can probably just double-click or click the downloaded file to unpack it.

Open a terminal window. This will be "Terminal" on OSX and "Git Bash" on Windows. You should see a black (or white) text window with a dollar sign in it.

Try typing

```
$ cd Downloads
$ unzip filesystem.zip
```

Once you have Nelle's files and directories, change to Nelle's home directory to begin, by typing in the `cd` command:

```
$ cd ~/Downloads/filesystem/users/nelle
```

Don't worry if you don't know what this command means yet! We will cover it soon.

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
nelle
$
```

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 is our current default directory, i.e., the directory that the computer assumes we want to run commands in unless we explicitly specify something else. Here, the computer's response is `/users/nelle`, which is Nelle's home directory:

```
$ pwd
/users/nelle/Downloads/filesystem/users/nelle
$
```

To understand what a “home directory” is, let’s have a look at how the file system as a whole is organized. At the top is the root directory that holds everything else. We refer to it using a slash character `/` on its own; this is the leading slash in `/users/nelle`.

Inside that directory are several other directories: `bin` (which is where some built-in programs are stored), `data` (for miscellaneous data files), `users` (where users’ personal directories are located), `tmp` (for temporary files that don’t need to be stored long-term), and so on:

We know that our current working directory `/users/nelle` is stored inside `/users` because `/users` is the first part of its name. Similarly, we know that `/users` is stored inside the root directory `/` because its name begins with `/`.

Let’s see what’s in Nelle’s home directory by running `ls`, which stands for “listing”:

```
$ ls
```

```
creatures  molecules      pizza.cfg
data       north-pacific-gyre  solar.pdf
Desktop    notes.txt       writing
```

`ls` prints the names of the files and directories in the current directory in alphabetical order, arranged neatly into columns. We can make its output more comprehensible by using the flag `-F`, which tells `ls` to add a trailing `/` to the names of directories:

```
$ ls -F
```

```
creatures/  molecules/      pizza.cfg
data/       north-pacific-gyre/  solar.pdf
Desktop/    notes.txt       writing/
```

Here, we can see that `filesystem/users/nelle` contains seven sub-directories. The names that don’t have trailing slashes, like `notes.txt`, `pizza.cfg`, and `solar.pdf`, are plain old files. And note that there is a space between `ls` and `-F`: without it, the shell thinks we’re trying to run a command called `ls-F`, which doesn’t exist.

3.5.1 What’s In A Name?

You may have noticed that all of Nelle’s files’ names are “something dot something”. This is just a convention: we can call a file `mythesis` or almost anything else we want. However, most people use two-part names most of the time to help them (and their programs) tell different kinds of files apart. The second part of such a name is called the filename extension, and indicates what type of data the file holds: `.txt` signals a plain text file, `.pdf` indicates a PDF document, `.cfg` is a configuration file full of parameters for some program or other, and so on.

This is just a convention, albeit an important one. Files contain bytes: it’s up to us and our programs to interpret those bytes according to the rules for PDF documents, images, and so on.

Naming a PNG image of a whale as `whale.mp3` doesn’t somehow magically turn it into a recording of whalesong, though it *might* cause the operating system to try to open it with a music player when someone double-clicks it.

Now let’s take a look at what’s in Nelle’s `data` directory by running `ls -F data`, i.e., the command `ls` with the arguments `-F` and `data`. The second argument—the one *without* a leading dash—tells `ls` that we want a listing of something other than our current working directory:

```
$ ls -F data
```

```
amino-acids.txt  elements/      morse.txt
pdb/             planets.txt    sunspot.txt
```

The output shows us that there are four text files and two sub-sub-directories. Organizing things hierarchically in this way helps us keep track of our work: it's possible to put hundreds of files in our home directory, just as it's possible to pile hundreds of printed papers on our desk, but it's a self-defeating strategy.

Notice, by the way that we spelled the directory name `data`. It doesn't have a trailing slash: that's added to directory names by `ls` when we use the `-F` flag to help us tell things apart. And it doesn't begin with a slash because it's a relative path, i.e., it tells `ls` how to find something from where we are, rather than from the root of the file system.

If we run `ls -F /data` (with a leading slash) we get a different answer, because `/data` is an absolute path:

```
$ ls -F /data
access.log    backup/      hardware.cfg
network.cfg
```

The leading `/` tells the computer to follow the path from the root of the filesystem, so it always refers to exactly one directory, no matter where we are when we run the command.

What if we want to change our current working directory? Before we do this, `pwd` shows us that we're in `filesystem/users/nelle`, and `ls` without any arguments shows us that directory's contents:

```
$ pwd
/Users/SWC/Downloads/filesystem/users/nelle

$ ls
creatures  molecules      pizza.cfg
data       north-pacific-gyre  solar.pdf
Desktop    notes.txt       writing
```

We can use `cd` followed by a directory name to change our working directory. `cd` stands for "change directory", which is a bit misleading: the command doesn't change the directory, it changes the shell's idea of what directory we are in.

```
$ cd data

cd doesn't print anything, but if we run pwd after it, we can see that we are now in
/Users/SWC/Downloads/filesystem/users/nelle/data. If we run ls without arguments now,
it lists the contents of /Users/SWC/Downloads/filesystem/users/nelle/data, because that's where
we now are:
```

```
$ pwd
/Users/SWC/Downloads/filesystem/users/nelle/data

$ ls -F
amino-acids.txt  elements/      morse.txt
pdb/             planets.txt    sunspot.txt
```

We now know how to go down the directory tree: how do we go up? We could use an absolute path:

```
$ cd /users/nelle
```

but it's almost always simpler to use `cd ..` to go up one level:

```
$ pwd
```

```
/users/nelle/data
```

```
$ cd ..
```

`..` is a special directory name meaning “the directory containing this one”, or more succinctly, the parent of the current directory. Sure enough, if we run `pwd` after running `cd ..`, we’re back in `/users/nelle`:

```
$ pwd
```

```
/users/nelle
```

The special directory `..` doesn’t usually show up when we run `ls`. If we want to display it, we can give `ls` the `-a` flag:

```
$ ls -F -a
```

```
./          Desktop/          pizza.cfg
../         molecules/       solar.pdf
creatures/  north-pacific-gyre/    writing/
data/      notes.txt
```

`-a` stands for “show all”; it forces `ls` to show us file and directory names that begin with `.`, such as `..` (which, if we’re in `/users/nelle`, refers to the `/users` directory). As you can see, it also displays another special directory that’s just called `.`, which means “the current working directory”. It may seem redundant to have a name for it, but we’ll see some uses for it soon.

3.5.2 Orthogonality

The special names `.` and `..` don’t belong to `ls`; they are interpreted the same way by every program. For example, if we are in `filesystem/users/nelle/data`, the command `ls ..` will give us a listing of `filesystem/users/nelle`. When the meanings of the parts are the same no matter how they’re combined, programmers say they are orthogonal: Orthogonal systems tend to be easier for people to learn because there are fewer special cases and exceptions to keep track of.

Nelle’s Pipeline: Organizing Files

Knowing just this much about files and directories, Nelle is ready to organize the files that the protein assay machine will create. First, she creates a directory called `north-pacific-gyre` (to remind herself where the data came from). Inside that, she creates a directory called `2012-07-03`, which is the date she started processing the samples. She used to use names like `conference-paper` and `revised-results`, but she found them hard to understand after a couple of years. (The final straw was when she found herself creating a directory called `revised-revised-results-3`.)

Nelle names her directories “year-month-day”, with leading zeroes for months and days, because the shell displays file and directory names in alphabetical order. If she used month names, December would come before July; if she didn’t use leading zeroes, November (‘11’) would come before July (‘7’). This is the purpose of [ISO standard 8601 Representation of dates and times](#)

Each of her physical samples is labelled according to her lab’s convention with a unique ten-character ID, such as “NENE01729A”. This is what she used in her collection log to record the location, time, depth, and other characteristics of the sample, so she decides to use it as part of each data file’s name. Since the assay machine’s output is plain text, she will call her files `NENE01729A.txt`, `NENE01812A.txt`, and so on. All 1520 files will go into the same directory.

If she is in her home directory, Nelle can see what files she has using the command:

```
$ ls north-pacific-gyre/2012-07-03/
```

This is a lot to type, but she can let the shell do most of the work. If she types:

```
$ ls nor
```

and then presses tab, the shell automatically completes the directory name for her:

```
$ ls north-pacific-gyre/
```

If she presses tab again, Bash will add `2012-07-03/` to the command, since it's the only possible completion. Pressing tab again does nothing, since there are 1520 possibilities; pressing tab twice brings up a list of all the files, and so on. This is called tab completion, and we will see it in many other tools as we go on.

3.6 Key Points

- The file system is responsible for managing information on the disk.
- Information is stored in files, which are stored in directories (folders).
- Directories can also store other directories, which forms a directory tree.
- `/` on its own is the root directory of the whole filesystem.
- A relative path specifies a location starting from the current location.
- An absolute path specifies a location from the root of the filesystem.
- Directory names in a path are separated with `'/'` on Unix, but `'\'` on Windows.
- `..` means “the directory above the current one”; `.` on its own means “the current directory”.
- Most files' names are `something.extension`. The extension isn't required, and doesn't guarantee anything, but is normally used to indicate the type of data in the file.
- Most commands take options (flags) which begin with a `'-'`.

3.7 Exercises

If `pwd` displays `/users/thing`, what will `ls ../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`

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`
4. Either #2 or #3 above, but not #1.

What does the command `cd` without a directory name do?

1. It has no effect.
2. It changes the working directory to `/`.
3. It changes the working directory to the user's home directory.
4. It produces an error message.

What does the command `ls` do when used with the `-s` and `-h` arguments?

Creating Things

4.1 Objectives

- Create a directory hierarchy that matches a given diagram.
- Create files in that hierarchy using an editor or by copying and renaming existing files.
- Display the contents of a directory using the command line.
- Delete specified files and/or directories.

4.2 Making Directories

We now know how to explore files and directories, but how do we create them in the first place? Let's go back to Nelle's home directory, `filesystem/users/nelle`, and use `ls -F` to see what it contains:

```
$ pwd

/Users/SWC/Downloads/filesystem/users/nelle

$ ls -F

creatures/  molecules/      pizza.cfg
data/       north-pacific-gyre/  solar.pdf
Desktop/    notes.txt        writing/
```

Let's create a new directory called `thesis` using the command `mkdir thesis` (which has no output):

```
$ mkdir thesis
```

As you might (or might not) guess from its name, `mkdir` means “make directory”. Since `thesis` is a relative path (i.e., doesn't have a leading slash), the new directory is made below the current working directory:

```
$ ls -F

creatures/  north-pacific-gyre/  thesis/
data/       notes.txt            writing/
Desktop/    pizza.cfg
molecules/  solar.pdf
```

However, there's nothing in it yet:

```
$ ls -F thesis
```

4.3 Let's make a file

Let's change our working directory to `thesis` using `cd`, then run a text editor called Nano to create a file called `draft.txt`:

```
$ cd thesis
$ nano draft.txt
```

4.4 Which Editor?

When we say, “nano is a text editor,” we really do mean “text”: it can only work with plain character data, not tables, images, or any other human-friendly media. We use it in examples because almost anyone can drive it anywhere without training, but please use something more powerful for real work. On Unix systems (such as Linux and Mac OS X), many programmers use [Emacs](#) or [Vim](#) (both of which are completely unintuitive, even by Unix standards), or a graphical editor such as [Gedit](#). On Windows, you may wish to use [Notepad++](#).

No matter what editor you use, you will need to know where it searches for and saves files. If you start it from the shell, it will (probably) use your current working directory as its default location. If you use your computer's start menu, it may want to save files in your desktop or documents directory instead. You can change this by navigating to another directory the first time you “Save As...”

Let's type in a few lines of text, then use Control-O to write our data to disk:

```
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
```

Once our file is saved, we can use Control-X to quit the editor and return to the shell. (Unix documentation often uses the shorthand `^A` to mean “control-A”.) `nano` doesn't leave any output on the screen after it exits, but `ls` now shows that we have created a file called `draft.txt`:

```
$ ls
```

```
draft.txt
```

Let's tidy up by running `rm draft.txt`:

```
$ rm draft.txt
```

This command removes files (“rm” is short for “remove”). If we run `ls` again, its output is empty once more, which tells us that our file is gone:

```
$ ls
```

4.5 Deleting Is Forever

Unix doesn't have a trash bin: when we delete files, they are unhooked from the file system so that their storage space on disk can be recycled. Tools for finding and recovering deleted files do exist, but there's no guarantee they'll work in any particular situation, since the computer may recycle the file's disk space right away.

Let's re-create that file and then move up one directory to `filesystem/users/nelle` using `cd ..`:

```
$ pwd

/Users/SWC/Download/filesystem/users/nelle/thesis

$ nano draft.txt
$ ls

draft.txt

$ cd ..
```

If we try to remove the entire `thesis` directory using `rm thesis`, we get an error message:

```
$ rm thesis

rm: cannot remove `thesis': Is a directory
```

This happens because `rm` only works on files, not directories. The right command is `rmdir`, which is short for “remove directory”. It doesn't work yet either, though, because the directory we're trying to remove isn't empty:

```
$ rmdir thesis

rmdir: failed to remove `thesis': Directory not empty
```

This little safety feature can save you a lot of grief, particularly if you are a bad typist. To really get rid of `thesis` we must first delete the file `draft.txt`:

```
$ rm thesis/draft.txt
```

The directory is now empty, so `rmdir` can delete it:

```
$ rmdir thesis
```

4.6 With Great Power Comes Great Responsibility

Removing the files in a directory just so that we can remove the directory quickly becomes tedious. Instead, we can use `rm` with the `-r` flag (which stands for “recursive”):

```
$ rm -r thesis # Warning! This can set you six years behind.
```

This removes everything in the directory, then the directory itself. If the directory contains sub-directories, `rm -r` does the same thing to them, and so on. It's very handy, but can do a lot of damage if used without care.

4.7 More Practice

Let's create that directory and file one more time. (Note that this time we're running `nano` with the path `thesis/draft.txt`, rather than going into the `thesis` directory and running `nano` on `draft.txt` there.)

```
$ pwd
```

```
/users/nelle
```

```
$ mkdir thesis
```

```
$ nano thesis/draft.txt
```

```
$ ls thesis
```

```
draft.txt
```

`draft.txt` isn't a particularly informative name, so let's change the file's name using `mv`, which is short for "move":

```
$ mv thesis/draft.txt thesis/quotes.txt
```

The first parameter tells `mv` what we're "moving", while the second is where it's to go. In this case, we're moving `thesis/draft.txt` to `thesis/quotes.txt`, which has the same effect as renaming the file. Sure enough, `ls` shows us that `thesis` now contains one file called `quotes.txt`:

```
$ ls thesis
```

```
quotes.txt
```

Just for the sake of inconsistency, `mv` also works on directories—there is no separate `mvdir` command.

Let's move `quotes.txt` into the current working directory. We use `mv` once again, but this time we'll just use the name of a directory as the second parameter to tell `mv` that we want to keep the filename, but put the file somewhere new. (This is why the command is called "move".) In this case, the directory name we use is the special directory name `.` that we mentioned earlier.

```
$ mv thesis/quotes.txt .
```

The effect is to move the file from the directory it was in to the current working directory. `ls` now shows us that `thesis` is empty:

```
$ ls thesis
```

Further, `ls` with a filename or directory name as a parameter only lists that file or directory. We can use this to see that `quotes.txt` is still in our current directory:

```
$ ls quotes.txt
```

```
quotes.txt
```

The `cp` command works very much like `mv`, except it copies a file instead of moving it. We can check that it did the right thing using `ls` with two paths as parameters—like most Unix commands, `ls` can be given thousands of paths at once:

```
$ cp quotes.txt thesis/quotations.txt
```

```
$ ls quotes.txt thesis/quotations.txt
```

```
quotes.txt  thesis/quotations.txt
```

To prove that we made a copy, let's delete the `quotes.txt` file in the current directory and then run that same `ls` again. This time it tells us that it can't find `quotes.txt` in the current directory, but it does find the copy in `thesis` that we didn't delete:

```
$ ls quotes.txt thesis/quotations.txt

ls: cannot access quotes.txt: No such file or directory
thesis/quotations.txt
```

4.8 Another Useful Abbreviation

The shell interprets the character `~` (tilde) at the start of a path to mean “the current user's home directory”. For example, if Nelle's home directory is `/home/nelle`, then `~/data` is equivalent to `/home/nelle/data`. This only works if it is the first character in the path: `here/there/~/elsewhere` is *not* `/home/nelle/elsewhere`.

4.9 Key Points

- Unix documentation uses ‘`^A`’ to mean “control-A”.
- The shell does not have a trash bin: once something is deleted, it's really gone.
- Nano is a very simple text editor—please use something else for real work.

4.10 Exercises

What is the output of the closing `ls` command in the sequence shown below?

```
$ pwd
/home/jamie/data
$ ls
proteins.dat
$ mkdir recombine
$ mv proteins.dat recombine
$ cp recombine/proteins.dat ../proteins-saved.dat
$ ls
```

Suppose that:

```
$ ls -F
analyzed/  fructose.dat    raw/    sucrose.dat
```

What command(s) could you run so that the commands below will produce the output shown?

```
$ ls
analyzed  raw
$ ls analyzed
fructose.dat  sucrose.dat
```

What does `cp` do when given several filenames and a directory name, as in:

```
$ mkdir backup
$ cp thesis/citations.txt thesis/quotations.txt backup
```

What does `cp` do when given three or more filenames, as in:

```
$ ls -F
intro.txt  methods.txt  survey.txt
$ cp intro.txt methods.txt survey.txt
```

The command `ls -R` lists the contents of directories recursively, i.e., lists their sub-directories, sub-sub-directories, and so on in alphabetical order at each level. The command `ls -t` lists things by time of last change, with most recently changed files or directories first. In what order does `ls -R -t` display things?

Pipes and Filters

5.1 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.
- Explain Unix's "small pieces, loosely joined" philosophy.

5.2 Multi-step operations with bash

Now that we know a few basic commands, we can finally look at the shell's most powerful feature: the ease with which it lets us combine existing programs in new ways. We'll start with a directory called `molecules` that contains six files describing some simple organic molecules. The `.pdb` extension indicates that these files are in Protein Data Bank format, a simple text format that specifies the type and position of each atom in the molecule.

```
$ ls molecules
```

```
cubane.pdb    ethane.pdb    methane.pdb
octane.pdb    pentane.pdb   propane.pdb
```

Let's go into that directory with `cd` and run the command `wc *.pdb`. `wc` is the "word count" command: it counts the number of lines, words, and characters in files. The `*` in `*.pdb` matches zero or more characters, so the shell turns `*.pdb` into a complete list of `.pdb` files:

```
$ cd molecules
```

```
$ wc *.pdb
```

```
20 156 1158 cubane.pdb
12  84  622 ethane.pdb
 9  57  422 methane.pdb
30 246 1828 octane.pdb
21 165 1226 pentane.pdb
15 111  825 propane.pdb
107 819 6081 total
```

5.3 Wildcards

`*` is a wildcard. It matches zero or more characters, so `*.pdb` matches `ethane.pdb`, `propane.pdb`, and so on. On the other hand, `p*.pdb` only matches `pentane.pdb` and `propane.pdb`, because the ‘p’ at the front only matches itself.

`?` is also a wildcard, but it only matches a single character. This means that `p?.pdb` matches `pi.pdb` or `p5.pdb`, but not `propane.pdb`. We can use any number of wildcards at a time: for example, `p*.p?*` matches anything that starts with a ‘p’ and ends with ‘.’, ‘p’, and at least one more character (since the ‘?’ has to match one character, and the final ‘*’ can match any number of characters). Thus, `p*.p?*` would match `preferred.practice`, and even `p.pi` (since the first ‘*’ can match no characters at all), but not `quality.practice` (doesn’t start with ‘p’) or `preferred.p` (there isn’t at least one character after the ‘p’).

When the shell sees a wildcard, it expands the wildcard to create a list of matching filenames *before* running the command that was asked for. This means that commands like `wc` and `ls` never see the wildcard characters, just what those wildcards matched. This is another example of orthogonal design.

If we run `wc -l` instead of just `wc`, the output shows only the number of lines per file:

```
$ wc -l *.pdb

 20  cubane.pdb
 12  ethane.pdb
  9  methane.pdb
 30  octane.pdb
 21  pentane.pdb
 15  propane.pdb
107  total
```

We can also use `-w` to get only the number of words, or `-c` to get only the number of characters.

Which of these files is shortest? It’s an easy question to answer when there are only six files, but what if there were 6000? Our first step toward a solution is to run the command:

```
$ wc -l *.pdb > lengths
```

The `>` tells the shell to redirect the command’s output to a file instead of printing it to the screen. The shell will create the file if it doesn’t exist, or overwrite the contents of that file if it does. (This is why there is no screen output: everything that `wc` would have printed has gone into the file `lengths` instead.) `ls lengths` confirms that the file exists:

```
$ ls lengths

lengths
```

We can now send the content of `lengths` to the screen using `cat lengths`. `cat` stands for “concatenate”: it prints the contents of files one after another. There’s only one file in this case, so `cat` just shows us what it contains:

```
$ cat lengths

 20  cubane.pdb
 12  ethane.pdb
  9  methane.pdb
 30  octane.pdb
 21  pentane.pdb
 15  propane.pdb
107  total
```


Now let's use the `sort` command to sort its contents. We will also use the `-n` flag to specify that the sort is numerical instead of alphabetical. This does *not* change the file; instead, it sends the sorted result to the screen:

```
$ sort -n lengths

 9  methane.pdb
12  ethane.pdb
15  propane.pdb
20  cubane.pdb
21  pentane.pdb
30  octane.pdb
107 total
```

We can put the sorted list of lines in another temporary file called `sorted-lengths` by putting `> sorted-lengths` after the command, just as we used `> lengths` to put the output of `wc` into `lengths`. Once we've done that, we can run another command called `head` to get the first few lines in `sorted-lengths`:

```
$ sort -n lengths > sorted-lengths
$ head -1 sorted-lengths

9  methane.pdb
```

Using the parameter `-1` with `head` tells it that we only want the first line of the file; `-20` would get the first 20, and so on. Since `sorted-lengths` contains the lengths of our files ordered from least to greatest, the output of `head` must be the file with the fewest lines.

If you think this is confusing, you're in good company: even once you understand what `wc`, `sort`, and `head` do, all those intermediate files make it hard to follow what's going on. We can make it easier to understand by running `sort` and `head` together:

```
$ sort -n lengths | head -1

9  methane.pdb
```

The vertical bar between the two commands is called a pipe. It tells the shell that we want to use the output of the command on the left as the input to the command on the right. The computer might create a temporary file if it needs to, or copy data from one program to the other in memory, or something else entirely; we don't have to know or care.

We can use another pipe to send the output of `wc` directly to `sort`, which then sends its output to `head`:

```
$ wc -l *.pdb | sort -n | head -1

9  methane.pdb
```

This is exactly like a mathematician nesting functions like $\sin(\pi x)^2$ and saying "the square of the sine of x times π ". In our case, the calculation is "head of sort of line count of `*.pdb`".

Here's what actually happens behind the scenes when we create a pipe. When a computer runs a program—any program—it creates a process in memory to hold the program's software and its current state. Every process has an input channel called standard input. (By this point, you may be surprised that the name is so memorable, but don't worry: most Unix programmers call it "stdin". Every process also has a default output channel called standard output (or "stdout").

The shell is actually just another program. Under normal circumstances, whatever we type on the keyboard is sent to the shell on its standard input, and whatever it produces on standard output is displayed on our screen. When we tell the shell to run a program, it creates a new process and temporarily sends whatever we type on our keyboard to that process's standard input, and whatever the process sends to standard output to the screen.

Here's what happens when we run `wc -l *.pdb > lengths`. The shell starts by telling the computer to create a new process to run the `wc` program. Since we've provided some filenames as parameters, `wc` reads from them instead

of from standard input. And since we've used `>` to redirect output to a file, the shell connects the process's standard output to that file.

If we run `wc -l *.pdb | sort -n` instead, the shell creates two processes (one for each process in the pipe) so that `wc` and `sort` run simultaneously. The standard output of `wc` is fed directly to the standard input of `sort`; since there's no redirection with `>`, `sort`'s output goes to the screen. And if we run `wc -l *.pdb | sort -n | head -1`, we get three processes with data flowing from the files, through `wc` to `sort`, and from `sort` through `head` to the screen.

This simple idea is why Unix has been so successful. Instead of creating enormous programs that try to do many different things, Unix programmers focus on creating lots of simple tools that each do one job well, and that work well with each other. This programming model is called pipes and filters. We've already seen pipes; a filter is a program like `wc` or `sort` that transforms a stream of input into a stream of output. Almost all of the standard Unix tools can work this way: unless told to do otherwise, they read from standard input, do something with what they've read, and write to standard output.

The key is that any program that reads lines of text from standard input and writes lines of text to standard output can be combined with every other program that behaves this way as well. You can *and should* write your programs this way so that you and other people can put those programs into pipes to multiply their power.

5.4 Downloading data with redirection

Nelle is frustrated and decides to download some 16S sequences to pass the time. She can download a collection of datafiles automatically. There is a list of datasets in `filesystem/data/metagenomics`

```
:: $ cd ~/Downloads/filesystem/data/metagenomics $ ls
datasetlist.sh generate_download_commands.sh
$
```

This command will download a dataset from the MG-RAST API and save it in `sequences.fasta`:

```
:: $ curl http://api.metagenomics.anl.gov/download/mgm4522007.3?file=050.1 > EbM1.fasta
```

If we examine this command line, we run the command “curl”; we give it a url with id numbers for the dataset we want, and the output of `curl` is redirected to the file `EbM1.fasta`.

`generate_download_commands.sh` uses columns 1 and 2 of the data table to construct a list of commands like that above to download the data from MG-RAST.

To automatically download the first five, we run `generate_download_commands.sh` and then run `download-volta-data.sh`:

```
:: $ bash generate_download_commands.sh Generating download-volta-data.sh $ bash download-volta-data.sh % Total % Received % Xferd Average Speed Time Time Time Current
      Dload Upload Total Spent Left Speed
```

5.4.1 Nelle's Pipeline: Checking Files

Nelle has run her samples through the assay machines and created 1520 files in the `north-pacific-gyre/2012-07-03` directory described earlier. As a quick sanity check, she types:

```
$ cd north-pacific-gyre/2012-07-03
$ wc -l *.txt
```

The output is 1520 lines that look like this:

```
300 NENE01729A.txt
300 NENE01729B.txt
300 NENE01736A.txt
300 NENE01751A.txt
300 NENE01751B.txt
300 NENE01812A.txt
... ..
```

Now she types this:

```
$ wc -l *.txt | sort -n | head -5
```

```
240 NENE02018B.txt
300 NENE01729A.txt
300 NENE01729B.txt
300 NENE01736A.txt
300 NENE01751A.txt
```

Whoops: one of the files is 60 lines shorter than the others. When she goes back and checks it, she sees that she did that assay at 8:00 on a Monday morning—someone was probably in using the machine on the weekend, and she forgot to reset it. Before re-running that sample, she checks to see if any files have too much data:

```
$ wc -l *.txt | sort -n | tail -5
```

```
300 NENE02040A.txt
300 NENE02040B.txt
300 NENE02040Z.txt
300 NENE02043A.txt
300 NENE02043B.txt
```

Those numbers look good—but what’s that ‘Z’ doing there in the third-to-last line? All of her samples should be marked ‘A’ or ‘B’; by convention, her lab uses ‘Z’ to indicate samples with missing information. To find others like it, she does this:

```
$ ls *Z.txt
```

```
NENE01971Z.txt    NENE02040Z.txt
```

Sure enough, when she checks the log on her laptop, there’s no depth recorded for either of those samples. Since it’s too late to get the information any other way, she must exclude those two files from her analysis. She could just delete them using `rm`, but there are actually some analyses she might do later where depth doesn’t matter, so instead, she’ll just be careful later on to select files using the wildcard expression `*[AB].txt`. As always, the ‘`*`’ matches any number of characters; the expression `[AB]` matches either an ‘A’ or a ‘B’, so this matches all the valid data files she has.

5.5 Key Points

- `command > file` redirects a command’s output to a file.
- `first | second` is a pipeline: the output of the first command is used as the input to the second.
- The best way to use the shell is to use pipes to combine simple single-purpose programs (filters).

5.6 Exercises

If we run `sort` on this file:

```
10
2
19
22
6
```

the output is:

```
10
19
2
22
6
```

If we run `sort -n` on the same input, we get this instead:

```
2
6
10
19
22
```

Explain why `-n` has this effect.

What is the difference between:

```
wc -l < mydata.dat
```

and:

```
wc -l mydata.dat
```

The command `uniq` removes adjacent duplicated lines from its input. For example, if a file `salmon.txt` contains:

```
coho
coho
steelhead
coho
steelhead
steelhead
```

then `uniq salmon.txt` produces:

```
coho
steelhead
coho
steelhead
```

Why do you think `uniq` only removes *adjacent* duplicated lines? (Hint: think about very large data sets.) What other command could you combine with it in a pipe to remove all duplicated lines?

A file called `animals.txt` contains the following data:

```
2012-11-05,deer
2012-11-05,rabbit
2012-11-05,raccoon
2012-11-06,rabbit
```

```
2012-11-06,deer
2012-11-06,fox
2012-11-07,rabbit
2012-11-07,bear
```

What text passes through each of the pipes and the final redirect in the pipeline below?

```
cat animals.txt | head -5 | tail -3 | sort -r > final.txt
```

The command:

```
$ cut -d , -f 2 animals.txt
```

produces the following output:

```
deer
rabbit
raccoon
rabbit
deer
fox
rabbit
bear
```

What other command(s) could be added to this in a pipeline to find out what animals the file contains (without any duplicates in their names)?

6.1 Objectives

- Write a loop that applies one or more commands separately to each file in a set of files.
- Trace the values taken on by a loop variable during execution of the loop.
- Explain the difference between a variable's name and its value.
- Explain why spaces and some punctuation characters shouldn't be used in files' names.
- Demonstrate how to see what commands have recently been executed.
- Re-run recently executed commands without retyping them.

Wildcards and tab completion are two ways to reduce typing (and typing mistakes). Another is to tell the shell to do something over and over again. Suppose we have several hundred genome data files named `basilisk.dat`, `unicorn.dat`, and so on. In this example, we'll use the `creatures` directory which only has two example files, but the principles can be applied to many many more files at once. When new files arrive, we'd like to rename the existing ones to `original-basilisk.dat` and `original-unicorn.dat`. We can't use:

```
$ mv *.dat original-*.dat
```

because that would expand (in the two-file case) to:

```
$ mv basilisk.dat unicorn.dat
```

This wouldn't back up our files: it would replace the content of `unicorn.dat` with whatever's in `basilisk.dat`.

Instead, we can use a loop to do some operation once for each thing in a list. Here's a simple example that displays the first three lines of each file in turn:

```
$ for filename in basilisk.dat unicorn.dat
> do
>   head -3 $filename
> done
```

```
COMMON NAME: basilisk
CLASSIFICATION: basiliscus vulgaris
UPDATED: 1745-05-02
COMMON NAME: unicorn
CLASSIFICATION: equus monoceros
UPDATED: 1738-11-24
```

When the shell sees the keyword `for`, it knows it is supposed to repeat a command (or group of commands) once for each thing in a list. In this case, the list is the two filenames. Each time through the loop, the name of the thing

currently being operated on is assigned to the variable called `filename`. Inside the loop, we get the variable's value by putting `$` in front of it: `$filename` is `basilisk.dat` the first time through the loop, `unicorn.dat` the second, and so on. Finally, the command that's actually being run is our old friend `head`, so this loop prints out the first three lines of each data file in turn.

The shell prompt changes from `$` to `>` and back again as we were typing in our loop. The second prompt, `>`, is different to remind us that we haven't finished typing a complete command yet.

We have called the variable in this loop `filename` in order to make its purpose clearer to human readers. The shell itself doesn't care what the variable is called; if we wrote this loop as:

```
for x in basilisk.dat unicorn.dat
do
    head -3 $x
done
```

or:

```
for temperature in basilisk.dat unicorn.dat
do
    head -3 $temperature
done
```

it would work exactly the same way. *Don't do this.* Programs are only useful if people can understand them, so meaningless names (like `x`) or misleading names (like `temperature`) increase the odds that the program won't do what its readers think it does.

Here's a slightly more complicated loop:

```
for filename in *.dat
do
    echo $filename
    head -100 $filename | tail -20
done
```

The shell starts by expanding `*.dat` to create the list of files it will process. The loop body then executes two commands for each of those files. The first, `echo`, just prints its command-line parameters to standard output. For example:

```
$ echo hello there
```

prints:

```
hello there
```

In this case, since the shell expands `$filename` to be the name of a file, `echo $filename` just prints the name of the file. Note that we can't write this as:

```
for filename in *.dat
do
    $filename
    head -100 $filename | tail -20
done
```

because then the first time through the loop, when `$filename` expanded to `basilisk.dat`, the shell would try to run `basilisk.dat` as a program. Finally, the `head` and `tail` combination selects lines 81-100 from whatever file is being processed.

Filename expansion in loops is another reason you should not use spaces in filenames. Suppose our data files are named:


```

basilisk.dat
red dragon.dat
unicorn.dat

```

If we try to process them using:

```

for filename in *.dat
do
    head -100 $filename | tail -20
done

```

then the shell will expand `*.dat` to create:

```

basilisk.dat red dragon.dat unicorn.dat

```

With older versions of Bash, or most other shells, `filename` will then be assigned the following values in turn:

```

basilisk.dat
red
dragon.dat
unicorn.dat

```

That's a problem: `head` can't read files called `red` and `dragon.dat` because they don't exist, and won't be asked to read the file `red dragon.dat`.

We can make our script a little bit more robust by quoting our use of the variable:

```

for filename in *.dat
do
    head -100 "$filename" | tail -20
done

```

but it's simpler just to avoid using spaces (or other special characters) in filenames.

Going back to our original file renaming problem, we can solve it using this loop:

```

for filename in *.dat
do
    mv $filename original-$filename
done

```

This loop runs the `mv` command once for each filename. The first time, when `$filename` expands to `basilisk.dat`, the shell executes:

```

mv basilisk.dat original-basilisk.dat

```

The second time, the command is:

```

mv unicorn.dat original-unicorn.dat

```

6.2 Measure Twice, Run Once

A loop is a way to do many things at once—or to make many mistakes at once if it does the wrong thing. One way to check what a loop *would* do is to echo the commands it would run instead of actually running them. For example, we could write our file renaming loop like this:

```
for filename in *.dat
do
    echo mv $filename original-$filename
done
```

Instead of running `mv`, this loop runs `echo`, which prints out:

```
mv basilisk.dat original-basilisk.dat
mv unicorn.dat original-unicorn.dat
```

without actually running those commands. We can then use up-arrow to redisplay the loop, back-arrow to get to the word `echo`, delete it, and then press “enter” to run the loop with the actual `mv` commands. This isn’t foolproof, but it’s a handy way to see what’s going to happen when you’re still learning how loops work.

6.2.1 Nelle’s Pipeline: Processing Files

Nelle is now ready to process her data files. Since she’s still learning how to use the shell, she decides to build up the required commands in stages. Her first step is to make sure that she can select the right files—remember, these are ones whose names end in ‘A’ or ‘B’, rather than ‘Z’:

```
$ cd north-pacific-gyre/2012-07-03
$ for datafile in *[AB].txt
> do
>     echo $datafile
> done
```

```
NENE01729A.txt
NENE01729B.txt
NENE01736A.txt
...
NENE02043A.txt
NENE02043B.txt
```

Her next step is to decide what to call the files that the `goostats` analysis program will create. Prefixing each input file’s name with “stats” seems simple, so she modifies her loop to do that:

```
$ for datafile in *[AB].txt
> do
>     echo $datafile stats-$datafile
> done
```

```
NENE01729A.txt stats-NENE01729A.txt
NENE01729B.txt stats-NENE01729B.txt
NENE01736A.txt stats-NENE01736A.txt
...
NENE02043A.txt stats-NENE02043A.txt
NENE02043B.txt stats-NENE02043B.txt
```

She hasn’t actually run `goostats` yet, but now she’s sure she can select the right files and generate the right output filenames.

Typing in commands over and over again is becoming tedious, though, and Nelle is worried about making mistakes, so instead of re-entering her loop, she presses the up arrow. In response, the shell redisplay the whole loop on one line (using semi-colons to separate the pieces):

```
$ for datafile in *[AB].txt; do echo $datafile stats-$datafile; done
```

Using the left arrow key, Nelle backs up and changes the command `echo` to `goostats`:

```
$ for datafile in *[AB].txt; do bash goostats $datafile stats-$datafile; done
```

When she presses enter, the shell runs the modified command. However, nothing appears to happen—there is no output. After a moment, Nelle realizes that since her script doesn’t print anything to the screen any longer, she has no idea whether it is running, much less how quickly. She kills the job by typing Control-C, uses up-arrow to repeat the command, and edits it to read:

```
$ for datafile in *[AB].txt; do echo $datafile; bash goostats $datafile stats-$datafile; done
```

```
Beginning and End
^^^^^^^^^^^^^^^^^^
```

We can move to the beginning of a line in the shell by typing ``^A`` (which means Control-A) and to the end using ``^E``.

When she runs her program now, it produces one line of output every five seconds or so:

```
NENE01729A.txt
NENE01729B.txt
NENE01736A.txt
...
```

1518 times 5 seconds, divided by 60, tells her that her script will take about two hours to run. As a final check, she opens another terminal window, goes into `north-pacific-gyre/2012-07-03`, and uses `cat stats-NENE01729B.txt` to examine one of the output files. It looks good, so she decides to get some coffee and catch up on her reading.

Another way to repeat previous work is to use the `history` command to get a list of the last few hundred commands that have been executed, and then to use `!123` (where “123” is replaced by the command number) to repeat one of those commands. For example, if Nelle types this:

```
$ history | tail -5
456  ls -l NENE0*.txt
457  rm stats-NENE01729B.txt.txt
458  bash goostats NENE01729B.txt stats-NENE01729B.txt
459  ls -l NENE0*.txt
460  history
```

then she can re-run `goostats` on `NENE01729B.txt` simply by typing `!458`.

6.3 Key Points

- A `for` loop repeats commands once for every thing in a list.
- Every `for` loop needs a variable to refer to the current “thing”.
- Use `$name` to expand a variable (i.e., get its value).
- Do not use spaces, quotes, or wildcard characters such as `*` or `?` in filenames, as it complicates variable expansion.
- Give files consistent names that are easy to match with wildcard patterns to make it easy to select them for looping.
- Use the up-arrow key to scroll up through previous commands to edit and repeat them.
- Use `history` to display recent commands, and `!number` to repeat a command by number.

6.4 Exercises

Suppose that `ls` initially displays:

```
fructose.dat    glucose.dat    sucrose.dat
```

What is the output of:

```
for datafile in *.dat
do
    ls *.dat
done
```

In the same directory, what is the effect of this loop?

```
for sugar in *.dat
do
    echo $sugar
    cat $sugar > xylose.dat
done
```

1. Prints `fructose.dat`, `glucose.dat`, and `sucrose.dat`, and copies `sucrose.dat` to create `xylose.dat`.
2. Prints `fructose.dat`, `glucose.dat`, and `sucrose.dat`, and concatenates all three files to create `xylose.dat`.
3. Prints `fructose.dat`, `glucose.dat`, `sucrose.dat`, and `xylose.dat`, and copies `sucrose.dat` to create `xylose.dat`.
4. None of the above.

The `expr` does simple arithmetic using command-line parameters:

```
$ expr 3 + 5
8
$ expr 30 / 5 - 2
4
```

Given this, what is the output of:

```
for left in 2 3
do
    for right in $left
    do
        expr $left + $right
    done
done
```

Describe in words what the following loop does.

```
for how in frogll prcb redig
do
    $how -limit 0.01 NENE01729B.txt
done
```

Shell Scripts

7.1 Objectives

- Write a shell script that runs a command or series of commands for a fixed set of files.
- Run a shell script from the command line.
- Write a shell script that operates on a set of files defined by the user on the command line.
- Create pipelines that include user-written shell scripts.

7.2 Now we're cooking

We are finally ready to see what makes the shell such a powerful programming environment. We are going to take the commands we repeat frequently and save them in files so that we can re-run all those operations again later by typing a single command. For historical reasons, a bunch of commands saved in a file is usually called a shell script, but make no mistake: these are actually small programs.

Let's start by going back to `molecules/` and putting the following line in the file `middle.sh`:

```
$ cd molecules
$ nano middle.sh
```

```
head -15 octane.pdb | tail -5
```

This is a variation on the pipe we constructed earlier: it selects lines 11-15 of the file `octane.pdb`. Remember, we are *not* running it as a command just yet: we are putting the commands in a file.

Once we have saved the file, we can ask the shell to execute the commands it contains. Our shell is called `bash`, so we run the following command:

```
$ bash middle.sh
```

```
ATOM      9  H          1      -4.502   0.681   0.785  1.00  0.00
ATOM     10  H          1      -5.254  -0.243  -0.537  1.00  0.00
ATOM     11  H          1      -4.357   1.252  -0.895  1.00  0.00
ATOM     12  H          1      -3.009  -0.741  -1.467  1.00  0.00
ATOM     13  H          1      -3.172  -1.337   0.206  1.00  0.00
```

Sure enough, our script's output is exactly what we would get if we ran that pipeline directly.

7.3 Text vs. Whatever

We usually call programs like Microsoft Word or LibreOffice Writer “text editors”, but we need to be a bit more careful when it comes to programming. By default, Microsoft Word uses `.docx` files to store not only text, but also formatting information about fonts, headings, and so on. This extra information isn’t stored as characters, and doesn’t mean anything to tools like `head`: they expect input files to contain nothing but the letters, digits, and punctuation on a standard computer keyboard. When editing programs, therefore, you must either use a plain text editor, or be careful to save files as plain text.

What if we want to select lines from an arbitrary file? We could edit `middle.sh` each time to change the filename, but that would probably take longer than just retyping the command. Instead, let’s edit `middle.sh` and replace `octane.pdb` with a special variable called `$1`:

```
$ cat middle.sh
```

```
head -20 $1 | tail -5
```

Inside a shell script, `$1` means “the first filename (or other parameter) on the command line”. We can now run our script like this:

```
$ bash middle.sh octane.pdb
```

```
ATOM      9  H          1      -4.502   0.681   0.785   1.00   0.00
ATOM     10  H          1      -5.254  -0.243  -0.537   1.00   0.00
ATOM     11  H          1      -4.357   1.252  -0.895   1.00   0.00
ATOM     12  H          1      -3.009  -0.741  -1.467   1.00   0.00
ATOM     13  H          1      -3.172  -1.337   0.206   1.00   0.00
```

or on a different file like this:

```
$ bash middle.sh pentane.pdb
```

```
ATOM      9  H          1       1.324   0.350  -1.332   1.00   0.00
ATOM     10  H          1       1.271   1.378   0.122   1.00   0.00
ATOM     11  H          1      -0.074  -0.384   1.288   1.00   0.00
ATOM     12  H          1      -0.048  -1.362  -0.205   1.00   0.00
ATOM     13  H          1      -1.183   0.500  -1.412   1.00   0.00
```

We still need to edit `middle.sh` each time we want to adjust the range of lines, though. Let’s fix that by using the special variables `$2` and `$3`:

```
$ cat middle.sh
```

```
head $2 $1 | tail $3
```

```
$ bash middle.sh pentane.pdb -20 -5
```

```
ATOM     14  H          1      -1.259   1.420   0.112   1.00   0.00
ATOM     15  H          1      -2.608  -0.407   1.130   1.00   0.00
ATOM     16  H          1      -2.540  -1.303  -0.404   1.00   0.00
ATOM     17  H          1      -3.393   0.254  -0.321   1.00   0.00
TER       18          1
```

This works, but it may take the next person who reads `middle.sh` a moment to figure out what it does. We can improve our script by adding some comments at the top:

```
$ cat middle.sh
```

```
# Select lines from the middle of a file.
# Usage: middle.sh filename -end_line -num_lines
head $2 $1 | tail $3
```

A comment starts with a # character and runs to the end of the line. The computer ignores comments, but they're invaluable for helping people understand and use scripts.

What if we want to process many files in a single pipeline? For example, if we want to sort our .pdb files by length, we would type:

```
$ wc -l *.pdb | sort -n
```

because `wc -l` lists the number of lines in the files and `sort -n` sorts things numerically. We could put this in a file, but then it would only ever sort a list of .pdb files in the current directory. If we want to be able to get a sorted list of other kinds of files, we need a way to get all those names into the script. We can't use \$1, \$2, and so on because we don't know how many files there are. Instead, we use the special variable \$*, which means, "All of the command-line parameters to the shell script." Here's an example:

```
$ cat sorted.sh

wc -l $* | sort -n

$ bash sorted.sh *.pdb ../creatures/*.dat

9 methane.pdb
12 ethane.pdb
15 propane.pdb
20 cubane.pdb
21 pentane.pdb
30 octane.pdb
163 ../creatures/basilisk.dat
163 ../creatures/unicorn.dat
```

7.4 Why Isn't It Doing Anything?

What happens if a script is supposed to process a bunch of files, but we don't give it any filenames? For example, what if we type:

```
$ bash sorted.sh
```

but don't say *.dat (or anything else)? In this case, \$* expands to nothing at all, so the pipeline inside the script is effectively:

```
wc -l | sort -n
```

Since it doesn't have any filenames, `wc` assumes it is supposed to process standard input, so it just sits there and waits for us to give it some data interactively. From the outside, though, all we see is it sitting there: the script doesn't appear to do anything.

We have two more things to do before we're finished with our simple shell scripts. If you look at a script like:

```
wc -l $* | sort -n
```

you can probably puzzle out what it does. On the other hand, if you look at this script:

```
# List files sorted by number of lines.
wc -l $* | sort -n
```

you don't have to puzzle it out—the comment at the top tells you what it does. A line or two of documentation like this make it much easier for other people (including your future self) to re-use your work. The only caveat is that each time you modify the script, you should check that the comment is still accurate: an explanation that sends the reader in the wrong direction is worse than none at all.

Second, suppose we have just run a series of commands that did something useful—for example, that created a graph we'd like to use in a paper. We'd like to be able to re-create the graph later if we need to, so we want to save the commands in a file. Instead of typing them in again (and potentially getting them wrong) we can do this:

```
$ history | tail -4 > redo-figure-3.sh
```

The file `redo-figure-3.sh` now contains:

```
297 goostats -r NENE01729B.txt stats-NENE01729B.txt
298 goodiff stats-NENE01729B.txt /data/validated/01729.txt > 01729-differences.txt
299 cut -d ',' -f 2-3 01729-differences.txt > 01729-time-series.txt
300 ygraph --format scatter --color bw --borders none 01729-time-series.txt figure-3.png
```

After a moment's work in an editor to remove the serial numbers on the commands, we have a completely accurate record of how we created that figure.

Nelle could also use `colrm` (short for “column removal”) to remove the serial numbers on her previous commands. Its parameters are the range of characters to strip from its input:

```
$ history | tail -5
173 cd /tmp
174 ls
175 mkdir bakup
176 mv bakup backup
177 history | tail -5
$ history | tail -5 | colrm 1 7
cd /tmp
ls
mkdir bakup
mv bakup backup
history | tail -5
history | tail -5 | colrm 1 7
```

In practice, most people develop shell scripts by running commands at the shell prompt a few times to make sure they're doing the right thing, then saving them in a file for re-use. This style of work allows people to recycle what they discover about their data and their workflow with one call to `history` and a bit of editing to clean up the output and save it as a shell script.

7.4.1 Nelle's Pipeline: Creating a Script

An off-hand comment from her supervisor has made Nelle realize that she should have provided a couple of extra parameters to `goostats` when she processed her files. This might have been a disaster if she had done all the analysis by hand, but thanks to for loops, it will only take a couple of hours to re-do.

But experience has taught her that if something needs to be done twice, it will probably need to be done a third or fourth time as well. She runs the editor and writes the following:

```
# Calculate reduced stats for data files at J = 100 c/bp.
for datafile in $*
do
```



```

    echo $datafile
    goostats -J 100 -r $datafile stats-$datafile
done

```

(The parameters `-J 100` and `-r` are the ones her supervisor said she should have used.) She saves this in a file called `do-stats.sh` so that she can now re-do the first stage of her analysis by typing:

```
$ bash do-stats.sh *[AB].txt
```

She can also do this:

```
$ bash do-stats.sh *[AB].txt | wc -l
```

so that the output is just the number of files processed rather than the names of the files that were processed.

One thing to note about Nelle's script is that it lets the person running it decide what files to process. She could have written it as:

```

# Calculate reduced stats for A and Site B data files at J = 100 c/bp.
for datafile in *[AB].txt
do
    echo $datafile
    goostats -J 100 -r $datafile stats-$datafile
done

```

The advantage is that this always selects the right files: she doesn't have to remember to exclude the 'Z' files. The disadvantage is that it *always* selects just those files—she can't run it on all files (including the 'Z' files), or on the 'G' or 'H' files her colleagues in Antarctica are producing, without editing the script. If she wanted to be more adventurous, she could modify her script to check for command-line parameters, and use `*[AB].txt` if none were provided. Of course, this introduces another tradeoff between flexibility and complexity.

7.5 Key Points

- Save commands in files (usually called shell scripts) for re-use.
- `bash filename` runs the commands saved in a file.
- `$*` refers to all of a shell script's command-line parameters.
- `$1`, `$2`, etc., refer to specified command-line parameters.
- Letting users decide what files to process is more flexible and more consistent with built-in Unix commands.

7.6 Exercises

Leah has several hundred data files, each of which is formatted like this:

```

2013-11-05,deer,5
2013-11-05,rabbit,22
2013-11-05,raccoon,7
2013-11-06,rabbit,19
2013-11-06,deer,2
2013-11-06,fox,1
2013-11-07,rabbit,18
2013-11-07,bear,1

```

Write a shell script called `species.sh` that takes any number of filenames as command-line parameters, and uses `cut`, `sort`, and `uniq` to print a list of the unique species appearing in each of those files separately.

Write a shell script called `longest.sh` that takes the name of a directory and a filename extension as its parameters, and prints out the name of the file with the most lines in that directory with that extension. For example:

```
$ bash longest.sh /tmp/data pdb
```

would print the name of the `.pdb` file in `/tmp/data` that has the most lines.

If you run the command:

```
history | tail -5 > recent.sh
```

the last command in the file is the `history` command itself, i.e., the shell has added `history` to the command log before actually running it. In fact, the shell *always* adds commands to the log before running them. Why do you think it does this?

Joel's data directory contains three files: `fructose.dat`, `glucose.dat`, and `sucrose.dat`. Explain what a script called `example.sh` would do when run as `bash example.sh *.dat` if it contained the following lines:

```
# Script 1
echo *.*

# Script 2
for filename in $1 $2 $3
do
    cat $filename
done

# Script 3
echo $*.dat
```

Getting started with Amazon EC2

8.1 Logging into your new instance “in the cloud”

OK, so we have rented a computer for you. It will be yours alone for the rest of the day, and it is a blank slate. It doesn't have blast (yet) and doesn't have any data.

If you are on windows and weren't here yesterday, you need to **install git bash**: [git bash](#).

The network name of your new computer we gave you on a scrap of paper.

The username for your instance will be “ubuntu”.

First, you must download the private key file from here: [soils.zip](#)

This gives us a compressed (and encrypted) zip file.

First, we need to uncompress it. You might be able to uncompress it automatically by clicking on it and entering the password, but:

```
cd ~/Downloads
unzip soils.zip
```

should be enough to give you soils.pem.

When you have soils.pem, move it onto your desktop.

Next, start Terminal (for mac) or gitbash (for windows) and type:

```
chmod og-rwx ~/Desktop/soils.pem
```

to set the permissions on the private key file to “closed to all evildoers”.

Then type:

```
ssh -i ~/Desktop/soils.pem ubuntu@ec2-???-???-???-???.compute-1.amazonaws.com
```

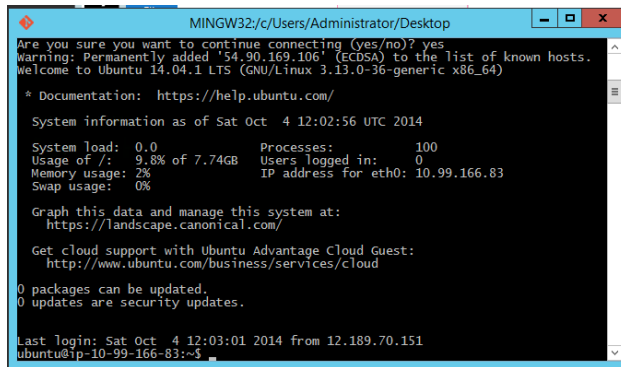
(Where you need to put the address for your machine, distributed on a sheet of paper, here)

This is the command that opens a bash connection to control your private instance in the cloud.

This command uses 'soils.pem' as the private key, uses ubuntu as the user name, and opens an encrypted connection and a bash shell on the 'ec2-???-???-???-???compute-1.amazonaws.com' server.

8.1.1 Declare victory

At the end you should see text and a prompt that look like this:



```
MINGW32/c/Users/Administrator/Desktop
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '54.90.169.106' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-36-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

System information as of Sat Oct  4 12:02:56 UTC 2014

System load:  0.0          Processes:    100
Usage of /:   9.8% of 7.74GB Users logged in:  0
Memory usage: 2%          IP address for eth0: 10.99.166.83
Swap usage:   0%

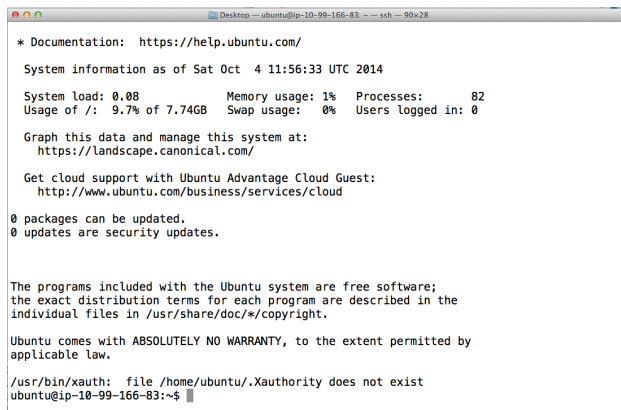
Graph this data and manage this system at:
https://landscape.canonical.com/

Get cloud support with Ubuntu Advantage Cloud Guest:
http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.

Last login: Sat Oct  4 12:03:01 2014 from 12.189.70.151
ubuntu@ip-10-99-166-83:~$
```

or, on mac:



```
Desktop - ubuntu@ip-10-99-166-83: ~ - ssh - 80x28

 * Documentation:  https://help.ubuntu.com/

System information as of Sat Oct  4 11:56:33 UTC 2014

System load:  0.08         Memory usage: 1%   Processes:    82
Usage of /:   9.7% of 7.74GB Swap usage:  0%   Users logged in: 0

Graph this data and manage this system at:
https://landscape.canonical.com/

Get cloud support with Ubuntu Advantage Cloud Guest:
http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

/usr/bin/xauth:  file /home/ubuntu/.Xauthority does not exist
ubuntu@ip-10-99-166-83:~$
```

Congratulations, you now have CONTROL of your own EC2 server.

We will keep these servers running for the rest of the day; if you want one for longer than that, you need \$.06/hour willingness to make it happen. (Think about it.)

8.1.2 Amazon Web Services reference material

You can find help online configuring, managing, and otherwise using EC2's interface.

- [Instance types](#)
- [Instance costs](#)

For this workshop, we provided 40 instances that were blank except for the `soils.pem` key.

If you would like to learn about instances on your own, try the original [2013 CEMI workshop at Caltech workshop](#), which includes instructions on how to start instances from scratch.

8.2 Amazon Web Services reference material

You can find help online configuring, managing, and otherwise using EC2's interface.

[Instance types](#) [Instance costs](#)

You can find a more detailed walkthrough (with more steps!) at the [2013 CEMI workshop at Caltech](#).

Basic EC2, command line, and BLAST

Log into your cloud instance machine with SSH (described in *Getting started with Amazon EC2*) and we can start to

- get the tools
- get the data
- get the comparison data
- run the tools
- get the data off

9.1 Install BLAST and some other software

You should be starting at the prompt that looks something like `'ubuntu@ip-10-98-166-81:~$'`, inside your Terminal or gitbash window.

This is now not bash on your local machine, but bash on amazon's cloud instance.

The instance we've started for you is a blank slate. It has bash, but little else. No blast, no genomes, no bwa...

Now that we are in the cloud, download and install BLAST:

```
curl -O ftp://ftp.ncbi.nih.gov/blast/executables/release/2.2.26/blast-2.2.26-x64-linux.tar.gz
tar xzf blast-2.2.26-x64-linux.tar.gz
sudo cp blast-2.2.26/bin/* /usr/local/bin
sudo cp -r blast-2.2.26/data /usr/local/blast-data
```

Download and install some useful scripts::

```
sudo apt-get -y install git python-pip
sudo git clone https://github.com/ngs-docs/ngs-scripts /usr/local/share/ngs-scripts
```

Create a working directory on a large disk, and change to that working directory:

```
cd /mnt
sudo chown $USER /mnt
mkdir blast
cd blast
```

Download the E. coli MG1655 protein data set:

```
curl -O http://ftp.ncbi.nlm.nih.gov/genomes/Bacteria/Escherichia_coli_K_12_substr__MG1655_uid57779/NO
```

This grabs that URL and saves the contents of 'NC_000913.faa' to the local disk.

Grab a Prokka-generated set of proteins

```
curl -O http://athyra.idyll.org/~t/ecoli0104.faa
```

Let's take a quick look at these files:

```
head ecoli0104.faa
head NC_000913.faa
```

Format it for BLAST and run BLAST of the O104 protein set against the MG1655 protein set:

```
formatdb -i NC_000913.faa -o T -p T
blastall -i ecoli0104.faa -d NC_000913.faa -p blastp -e 1e-12 -o 0104.x.NC
```

Look at the output file:

```
head 0104.x.NC
```

Let's convert 'em to a CSV file:

```
sudo pip install screed
python /usr/local/share/ngs-scripts/blast/blast-to-csv-with-names.py ecoli0104.faa NC_000913.faa 0104.x.NC
```

This creates a file '0104.x.NC.csv', which you can open in Excel.

The notes from the Berkeley workshop show you how to synchronize high-value, output files on your instance using Dropbox. This tutorial does not show off this neat feature.

But we will copy the spreadsheet onto your computer using scp.

In a new Terminal or gitbash window (one that is NOT running SSH into your instance), type:

```
scp -i ~/Desktop/soils.pem ubuntu@ec2-???-???-???-???-compute-1.amazonaws.com:/mnt/blast/0104.x.NC.csv
```

(Note: there is a period at the end of this line. It is essential. scp **USER@HOST:DIRECTORY/FILENAME DESTINATION**)

(Why does it need to be a new terminal window? Your first terminal window is busy running SSH into the cloud instance, and your cloud instance is not authorized to write files to your laptop.)

This will copy the output file 0104.x.NC.csv to your current directory where you can open it, say, with excel.

9.1.1 Reciprocal BLAST calculation

Do the reciprocal BLAST, too:

```
formatdb -i ecoli0104.faa -o T -p T
blastall -i NC_000913.faa -d ecoli0104.faa -p blastp -e 1e-12 -o NC.x.0104
```

Extract reciprocal best hit:

```
python /usr/local/share/ngs-scripts/blast/blast-to-ortho-csv.py ecoli0104.faa NC_000913.faa 0104.x.NC.csv
```

This generates a file 'ortho.csv', containing the ortholog assignments and their annotations.:

```
scp -i ~/Desktop/soils.pem ubuntu@ec2-???-???-???-???-compute-1.amazonaws.com:/mnt/blast/ortho.csv
```

9.2 A few post-tutorial links

Explore the NCBI bacterial genome site here: <http://ftp.ncbi.nlm.nih.gov/genomes/Bacteria>

- ‘.faa’ files are protein data sets;
- ‘.fna’ files are genomic DNA;
- the rest are annotation files of various kinds.

Mapping with bwa

One of the most common operations when dealing with NGS data is mapping sequences to other sequences, and most commonly, mapping reads to a reference. Because short read alignment is so common, there is a deluge of programs available for doing it, and it is well-understood problem. Most of the efficient programs work by first building an *index* of the sequence that will be mapped to, which allows for extremely fast lookups, and then running a separate module which uses that index to align short sequences against the reference. Alignment is used for visualization, coverage estimation, SNP calling, expression analysis, and a slew of other problems. For a high-level overview, try [this NCBI review](#).

10.1 Getting the Dependencies

bwa is one of the many available fast read mappers that uses the Burrows-Wheeler transform to speed up finding exact and near-exact matches in the database. A review can be found [here](#).

The [original verison of this lesson](#) had instructions to download, uncompress, make, and install bwa.

We can get bwa on ubuntu with

```
sudo apt-get install bwa
```

This may not get us the most recent version of bwa, but the last version that was bundled for ubuntu. Now we have bwa, and we didn't have to brave a grumpy sysadmin to install it for us. On the cloud, you are your own sysadmin.

10.2 Getting the Data

First, we will create a new directory to contain our bwa alignments (separate from our blast alignments in the previous lesson):

```
cd /mnt
mkdir ecoli
cd ecoli
```

We'll be using the data we downloaded during the reads and quality control session; if you missed that, you'll eventually want to run through it in `reads_and_qc`, but for now, you can just grab the quality-controlled data with:

```
curl -O https://s3.amazonaws.com/public.ged.msu.edu/SRR390202.pe.qc.fq.gz
curl -O https://s3.amazonaws.com/public.ged.msu.edu/SRR390202.se.qc.fq.gz
```

This download takes about ten minutes to put on our instances.

You'll also need a reference genome, which can be acquired with:

```
curl -O http://ftp.ncbi.nlm.nih.gov/genomes/Bacteria/Escherichia_coli_O104_H4_2011C_3493_uid176127/NC
```

which downloads the reference genome from NCBI.

10.3 Mapping the Reads

To speed up the demonstration, we will just map a subset of the reads rather than the entire file, which is somewhat large (though small compared to many datasets). The head command outputs the first n lines of a file, by default 4:

```
gunzip -c ../SRR390202.pe.qc.fq.gz | head -400000 > ecoli_pe.fq
gunzip -c ../SRR390202.se.qc.fq.gz | head -400000 > ecoli_se.fq
```

We've got our reads and a reference, so we're ready to get started. First, we build an index of the reference genome using bwa:

```
mv ../NC_018658.fna .
bwa index -a bwtsv NC_018658.fna
```

The -a flag tells bwa which indexing algorithm to use. The program will automatically output some files with set extensions, which the main alignment program knows the format of. Thus, we run the alignment like so:

```
bwa mem -p NC_018658.fna ecoli_pe.fq > aln.x.ecoli_NC_018658.sam
```

which aligns the left and right reads against the reference, and outputs them to the given SAM file. SAM is a common format for alignments which is understood by many programs, along with BAM. It's often useful to have both, so we'll use a utility called samtools to produce a sorted BAM file as well. First, install samtools:

```
cd /mnt
curl -O -L http://sourceforge.net/projects/samtools/files/samtools/0.1.18/samtools-0.1.18.tar.bz2
tar xvfj samtools-0.1.18.tar.bz2
cd samtools-0.1.18
make
cp samtools /usr/local/bin
cd misc/
cp *.pl maq2sam-long maq2sam-short md5fa md5sum-lite wgsim /usr/local/bin/
cd ..
cd bcftools
cp *.pl bcftools /usr/local/bin/
```

Then, run samtools to do the conversion:

```
cd /mnt/ecoli
samtools view -uS aln.x.ecoli_NC_018658.sam > aln.x.ecoli_NC_018658.bam
samtools sort aln.x.ecoli_NC_018658.bam aln.x.ecoli_NC_018658.bam.sorted
samtools index aln.x.ecoli_NC_018658.bam.sorted.bam
```

For additional resources on these tools, check out:

- the [bwa manual](#)
- [info on samtools](#)
- the [SAM format spec](#)

10.4 Visualizing your Data with Tablet

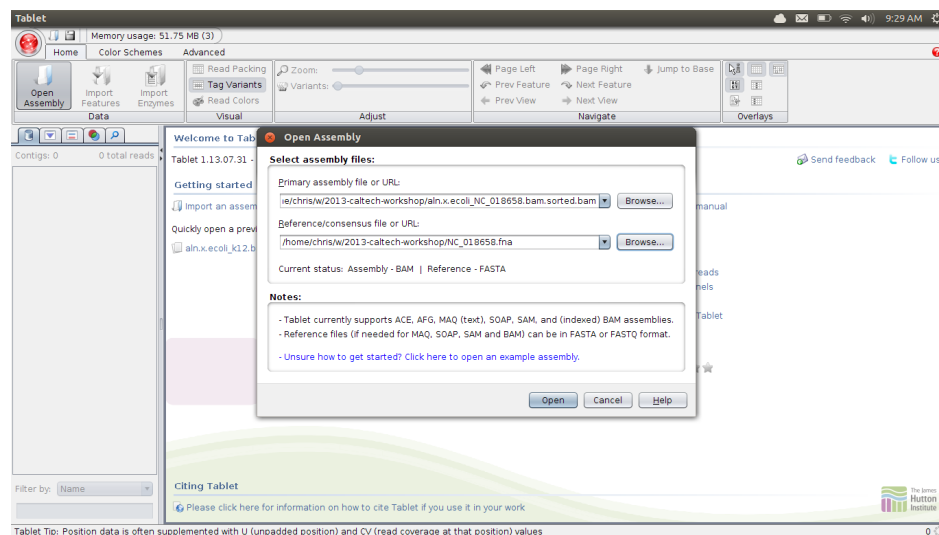
Copy your mapping files to your local machine :

```
: scp -i soils.pem ubuntu@yourinstance:/mnt/ecoli/aln.x.ecoli_NC_018658.bam.sorted.bam* .  scp -i soils.pem ubuntu@yourinstance:/mnt/ecoli/NC_018658.fna .
```

Although you can do many things with your alignments, one useful thing is to simply view them through a graphical interface. To demonstrate this, we'll use a program called Tablet, which can be downloaded [here](#).

Tablet claims to run on Windows, Linux, and OSX, though I have only tested it out on Linux. Because this is a GUI-driven program, we'll be running it on our local machines instead of our EC2 instances. So, go ahead and grab the appropriate version for your system, and install it.

Once you have it installed, open it up. You'll want to load your mapping file and reference genome:



You'll then get some loading bars, and potentially an error about the indexing file which can be ignored. You need to select a contig on the left to view; *ecoli* has a very good reference, and is only one contig:



You can move left and right along the contig, as well as zoom. Tablet can view other information like gene structure, but we won't get into that.