Getting Started in UNIX – Tutorial by Sheri Sanders (ss93@iu.edu)

Feel free to email me if you find errors, lack of clarity, etc. Thanks!

Where do I start?

****Getting onto a machine****

Most people tend to use either Mac OSX or Windows operating systems as opposed to some flavor of Linux. However, much of the software we use, and most of the university's computing resources, run on Linux (we have Red Hat Linux).

To save us the trouble of installing our own versions of Linux on our personal computers, and to gain access to the powerful machine and bioinformatics software used on IU's campus, we will learn how to do a remote login.

Remote login is the method of signing into and using a computer other than the one you are physically working on.

There are two basic methods for doing remote login: Putty and ssh.

1. PuTTY - if you are using Windows this is how you'll remotely log in. You'll need to install this free program called PuTTY. It's a "client" program that connects over a network to a "server," the machine we're trying to access.

http://www.putty.org/, download the putty.exe file. 8

To sign in, type your netID, @, and the hostname is the name of the machine we're logging into (i.e. jsmith@karst.uits.iu.edu), and set the connection type is ssh. Set Port to 22. Click open, and you do this you will be prompted for your password. Your username and password are your id and password.

2. ssh - "secure shell" - if you have a Mac, this is the command you will want to use to connect to a remote machine.

First you'll need to open up Terminal (you can get to it through Spotlight; I'd put it on your dashboard). Second you'll need to issue the ssh command. The format is as follows:

ssh username@karst.uits.iu.edu

******In The Shell*******

What we mostly use every day on our computers is a graphical user interface (GUI) - we can click, highlight, drag, and in various other ways interact with graphical components that represent the contents of our computer.

There's a different way to interact, and that's via the shell (a command line interface, or CLI), where we can textually navigate the computer we're on. Everything you can do in the GUI you can do in the shell, though it's obviously not always as easy since you have to issue commands instead of just using a mouse. Think of the difference between a touch screen and a mouse; there's an extra level of distance between you and what you can do.

We're going to be working with a shell called bash. Some of you might see the similarities between this terminal and MS-DOS, if you remember that far back. As you can see I just have a little box with some stuff up here at the front and a little blinking cursor. As you'd expect, we interact with this by typing stuff since that's really all I can do. So let's try something. There's a command called echo that does pretty much what you'd expect - it echos what you tell it.

echo hello class!

Sweet, it echoed it for us. But why is that useful? Well, among other things, we can use it to show us the values of variables in the shell. A variable is just like a variable in math. We name it something, and assign a value to it, like x = 4. You can assign variables, but the computer also has some variables it has already assigned. Let's see what my current name is:

echo $USER

Shell variables start with a $ - so echo USER just gives me USER. I can see where my home folder is with $HOME.

echo $HOME

I mentioned that there are different shells. How do you know which one you're in? Well, you can check with a shell variable. $0 tells us the name of the shell we're running.

echo $0

Remember how I said you can declare your own shell variables? Let's see how to do that.

declare myvar=hello

Notice that there's no $ in there when we declare it. However, when we want to echo it we need to tell echo that this is a variable, not just the word myvar. Also, DON'T put a space between myvar and =, and = and hello. If you do, you'll get this:

-bash: declare: `=': not a valid identifier

This is because UNIX likes to use whitespace (spaces, tabs, etc) as delimiters for commands. We'll get back to this.

But for now let's check what we have:

echo $myvar

Now we have a variable! Not everything can be that easy though. If we want to do something like:

declare myvar=hello class

We run into a problem. Only the first word was put into the variable. I said that you need to make sure not have a space between the myvar and the =, and that's because spaces aren't just white space in the shell all the time - they can mean things. There are other characters too like slashes, dollar signs, and a host of others that mean things to the computer, so we need a way to tell it that we want the literal thing we typed. We can do this using the escape character, the \.

declare myvar=hello\ class

And now it works! Use the escape character BEFORE the character you want to use literally (such as before the space). In some cases we can also put quotes around things to make sure that spaces are taken literally. The escape character is still VERY important to know, though.

You can also make it work with single or double quotes:

declare myvar="hello all"

declare myvar='hello everybody'

Now, why do we have three ways of doing this? Because the rules on each are slightly different. For example, if we wanted to have the terminal echo $4 (four dollars), we can try the most common double quotes:

declare myvar="$4" echo $myvar

But it’s empty... That's because characters inside double quotes are interpreted, as in Unix takes the special characters, like spaces and variable defining $ to mean something. It is reporting what is stored in $4, which is currently nothing. This can be useful in situations when you want to report a variable in context:

```
declare myvar="My name is $USER and I am using $0"

echo $myvar
```

If you want it to be read literally, use single quotes:

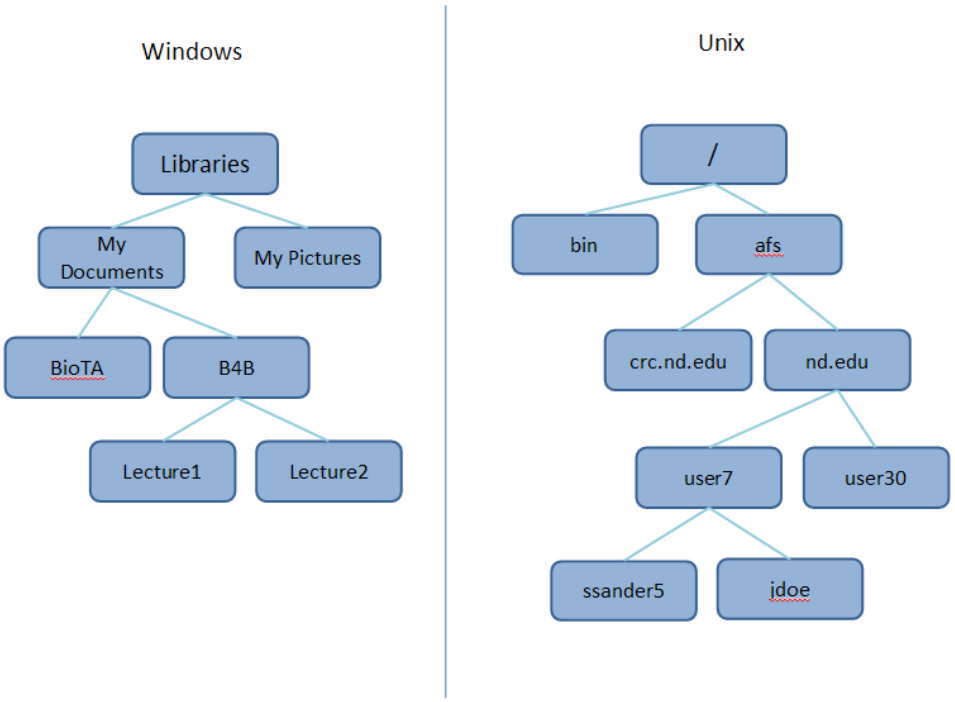
```
declare myvar='echo $USER will report your username'

echo $myvar
```

If you look at the attached Cheatsheet, there’s a special character list, and you’ll see here the difference between double and single quotes. I typically use double quotes because you can use variables (like $USER or $HOME) inside of them. Single quotes, however, will allow you to use an exclamation point or dollar sign without problem because it doesn’t let anything inside the quotes be interpreted. Since you probably won’t be using an exclamation point in your shell scripts that often, it’s pretty safe to use double quotes...but like I said before, just keep in mind that sometimes you have to fiddle with things to get them to work.

Introduction to the File System

Most file systems are hierarchical, like a tree. Below is how Windows and Unix are structured. Note Unix has a defined root of the tree (the most inclusive folder, the parent) is /.



Unlike Windows, Unix uses forward slashes to denote separations in folders in the file system. For example /afs/nd.edu/user7/ssander5 is my home directory on the ND computers, whereas ...My Documents\B4B\Lecture2 is a directory on my Windows machine.

NOTE: Where are things like CD-ROM drives, or portable hard drives we plug in, though? On Windows, we assign each of these things (including the disk drive of the machine itself) a letter, like the C: drive and the D: drive, and they have their own hierarchical tree. On Unix, these devices can be mounted anywhere in the main tree. For example, CD-ROM files could be found in /media/cdrom0. Where they end up in the tree is controlled by the administrator (root), so we won't worry about it. The point is it all looks like one big seamless file system, despite the fact that we have external things attached to it. It's kind of a Unix paradigm that everything can be treated like a file, including your entire CD drive.

******Flags and File Viewing******

If you want to look at your home directory, recall $HOME. 18

echo $HOME

We can also do this quicker with

echo ~

Ok, great. Let's look at something slightly different, and see how you see where you are at any

given point.

NOTE: My bash will look different from your bash for a little bit. In another week or so, we will adjust that for you!

You might want to see the entire path of your location (like the when we echo $HOME) even if you aren't at your home folder. You can view your current location by simply typing:

pwd

This command gives us our current working directory (**pwd stands for print working directory**. A directory is the same as a folder). You can see the full path now (which matches $HOME since that's where we are currently located).

When I echo $HOME you can see that this is my afs space. AFS is the distributed file system that ND uses. I'm not going to go into the particulars at all, but this is my afs space so it's not really on the individual machine I'm on, but on the network, and I can access it from any ND machine. This is why it doesn't matter which machine you log into.

Ok...but what's in my current directory? We can list the contents using what I think of as "let's see":

ls

Wonderful. I can see files, but I can't see all the wonderful information that I get on my Windows GUI, such as date of file creation, etc.

ls -l

There it is (sort of). The -l is called a "flag", it's an option in the command that we ran. It stands for "list" or "long", which lists all the file information. We will talk about weird dxr-'s next week.

Ok, so now I know I have options, but what other options do I have?? Enter: the man(ual) pages!

man ls

Look at all that wonderful information. Man pages give us lots of good info about the ls command, what it is, and its options. You can scroll down with the arrow keys (most programs will be like this), and you can exit just by typing q (this is also something of a standard so it's a good thing to try if you need to get out of something and you don't know how, Ctrl C is another

thing to try). One important thing to note is that case does matter. While this isn't always the case, it's important to typically try to treat things as case-sensitive, because, for example, -a and -A are not the same flag.

Notice that there are shortened options with one - and longer flag names with --. This is a legacy from when (incorrectly) typing command names was costly because terminals were very slow.

Anyway, let's try another common flag, -h, for human readable.

ls -l -h

Oh look, the file size is much more logical. We can also combine flags, if they are of the - variety.

ls -lh

Finally, the last really common "let's see" option -a, for all. Let's see a list of all the files in human readable format:

ls -lah

Look at all those files! Any file that starts with a "." is hidden from normal lists. This is akin to hidden administrative files on mac and windows and refers to administration files as well. A notable one that we will work with later is .bashrc, which is your bash profile that customizes your bash experience.

NOTE: Notice my pretty colors? That is a flag I have permanently turned on in my bash profile:

ls --color

Some more to check out, look at the man file if it's not obvious:

ls -r

ls -R

ls -t

******Navigating in the Shell******

Well, I don't want to stay in my home directory forever, so let's go adventuring. You can go somewhere else by **changing directories**. Let's go somewhere important, like... your scratch

space.

It's located:

/N/dc2/scratch/username

How do we get there?

Syntax: cd LOCATION

$cd /N/dc2/scratch/username

Ok, now we are in the course folder. Let's see what's in here!

ls -lah --color

Ok, let's make our own folder, with the make directory command:

Syntax: mkdir NAME

mkdir NewData
ls

A quick aside about directory naming: Earlier we talked about how spaces mean things, and thus we need to escape them when we want to use them. For this reason, I try to avoid giving folders names that have spaces in them. It's super straightforward and easy to mess with in the GUI, but when you're using the command line it gets annoying having to type stuff like cd This\ Folder when ThisFolder is just as readable and much less annoying to type.

Another reason not to do spaces? If you use :

mkdir All the things
ls –lah

You will see that there are three new folders. Because spaces separate commands in Unix. You are telling it to make three folders. If you instead use:

mkdir "All the things" or mkdir All\ the\ things

You will get a folder and enjoy the very annoying experience of having to escape those spaces every time you refer to the folder.

You can see from the blue text, as well as the d all the way to the left, what are directories (also

apparent if you use –F) and what are files. We also see two directories at the top, "." and "..". Let's investigate...

```
cd .

pwd
```

...huh. That did nothing... ok, let's try the other one...

```
cd ..

pwd
```

...Ok, that brought us up one folder. These are two special references (like ~ is for HOME), that are shorthand for the folder you are in and the folder above. Can we use ~ the same way?

```
cd ~

pwd
```

Yup. And there is one more special character for cd, "-".

```
cd -

pwd
```

Review:

~ -> HOME
. -> present folder
.. -> parent folder
- -> previous folder

So... why do we want these symbols confusing us? Because you can type in locations one of two ways:

1) from root (absolute): cd /N/dc2/scratch/ss93/NewData
2) from where you are(relative): ../OldData

It is often much easier to do the latter.

There's one quick trick that you should use, tab complete. If you are typing a command, hit tab to complete the command to the point where more information is needed. For example – if we

had two directories called Example and ExtraExample, we could type “cd Ex(tab)” and nothing would happen – because both folders start with Ex. However, if we type “cd Ext(tab)”, it would fill in the WHOLE DIRECTORY NAME. Using tab complete will ensure that you are spelling things correctly and getting the path correct. If you hit tab and it doesn’t complete words you think it should... you did something wrong ^_^.

******Interacting with Files and Directories******

Now we can move around in the shell and see where we are, but we also need to be able to look at file contents, make directories, and move files around.

Let’s start with a basic way to look at the contents of a file. But first we have to make a file. There are various editors for command line - the most simplistic is nano, so I’ll show you how to use it. If you want to get adventurous you can look into emacs or vim (I don't), but they’re full of lots of commands and weird key combinations, so we’ll stick to the easy stuff.

To edit an existing file or create a new one is the same. We’ll make a new file called This.txt, but if This.txt already existed this command would open the file and display its contents for editing:

```
nano This.txt
```

Now we have a new file! I’m going to type some random stuff in it and then close it. To exit nano, hold down control and hit the x key (at the bottom the ^ is for control). It’ll ask if you want to save (hit y for yes), and then it’ll ask you the filename to write. Unless you want to save to a new file just hit enter here, as your filename that you opened is already in this space.

Now let’s read the file (won’t be able to edit it) to confirm it saved.

```
cd ~
less This.txt
```

Note that you can tab complete the name, which will ensure it is spelled correctly (I HIGHLY recommend tab completing everything). If this were longer I could scroll through with arrow keys like I did on ls’s man page. Like with man, you exit by typing q.

We can also move files:

Syntax: mv <SOURCE> <DESTINATION>

```
mv This.txt NewData
ls
```

We can also look in folders without moving:

```
ls NewData
```

Mv also renames things, renaming something is essentially ‘moving’ something to a new filename.

```
cd NewData
mv This.txt That.txt
ls
```

And now This.txt is renamed That.txt.

NOTE: Please for the sanity of all, use file extensions. It’s hard to know what something is if you don't manually label it.

We can also copy or remove things. Copying and removing directories is a little different since there’s stuff in them.

Syntax: cp <SOURCE> <DESTINATION>

```
cp That.txt Other.txt
ls
```

Sometimes directories (folder act different). If we try to move a folder, we get an error:

```
mkdir Test
cp Test Test2
```

How do you solve this?

```
man cp
```

-r is a flag that tells cp to be recursive (to access everything in the directory).

```
cd ..
cp -r Test Testing
```

Now how to remove files:

```
rm Copy.txt
```

Similar to cp, directories need a -r flag

rm -r Testing

Depending on permissions of the files, it might ask if you're sure you want to delete things. To make it not do this, you can force your remove to happen with the -f flag.

rm -rf Testing

If you're using rm be VERY certain that's what you want to do - there's no trash can. I've accidentally deleted things before and have been very, very sad.

If you are going to create a bunch of files, we should probably know how to make new folders as well. This is done with the mkdir command:

mkdir Stuff

A good thing to do, starting right now, is to get in the habit of making a file called README whenever you make a new major folder. Why this README file? README, in all caps, is a conventional file you will find in many folders, particularly if you downloaded it or it's a shared use folder. A VERY good habit to get into is to make one for any folder you use for the sanity of 1) your PI –who will be thrilled to know what these files are when you graduate, 2) your future self – who will be thrilled to know what AvB.redone.txt.filtered is and how it was redone and filtered, and 3) your colleagues/coauthors – who will be thrilled to know what any new "redone" files are floating around in the common space. You don't have to do it for every folder, but it is much better to overdo it than under do it.

nano README

Things that are good to put in READMEs:

*The purpose of this folder, analysis, etc
*The editions of software used to perform the analysis
*Any really useful code or troubleshooting you had to do
*Any naming conventions for files (MF is male fasciularis, FM is female macaque etc.)

******Moving files across computers******

There are many times when you will want to move a file to your computer from your cluster space or vice versa. There are two ways to do this, one that is command line and one that is just easier.

1) command line - secure copy

Secure copy, scp, allows you to copy files from one machine to another. It works much like cp and move, but you have to give it a little more information on where things are, because the assumption is that it's on the same computer unless told otherwise. This command is very similar to copy, but with the additional <DESTINATION COMPUTER> before a file that exists elsewhere.

Syntax: scp <Source> <Destination Computer>:<File Location>

scp myfile.txt ss93@karst.uits.iu.edu:

Note the : with nothing after it in the example. The default is to copy files into your home folder. If you wish to copy it elsewhere, you can also do that by putting the location after the colon.

scp myfile.txt ss93@karst.uits.iu.edu:samples/

Additionally, you can copy from the cluster to your current machine:

scp ss93@karst.uits.iu.edu:samples/samplefile.txt .

It's still the same syntax - you are specifying where the file is and where it is going. And by the way, you cannot use you have to log into the other computer (which is why you use the <username>@<computer> and have to put in a password). If you've watched the sign in video, you know you can't sign into a computer from itself.

2) FTP - Filezilla

Unfortunately, this is not always easy to do, especially if you are on a Windows machine or if you have weird firewall/security issues going on. Most biologists I know prefer to instead use FTP or Filezilla.

Obtain and install Filezilla from: https://filezilla-project.org/

WARNING: DO NOT DO THE DEFAULT INSTALLATION. It's not a great idea for anything really. Make sure you do the custom installation and unselect anything that is not Filezilla.

Filezilla connects to the remote computers in a similar fashion to Putty or ssh - you specify your username, host computer, password, and port and it will connect to the remote computer. You can then access the files on the right side (remote site), or left side (local site).

******A Few Last Useful Things******

I'll give you two more quick tips to help you navigate on the command line. First, if you want to

reissue a command you've already done, like using less again. If you hit the up arrow key you can go back through your previous commands in order. Second, you can use wildcards in commands. I'll show you with ls.

ls file*.txt
ls file?.txt
ls file??.txt

* and ? are both special characters (like space), so if you want them taken literally you'll need to escape them with \. * says there can be any number of characters between file and .txt (including none). Each ? represents a single wildcard character, so the second will find something called files.txt but not files1.txt. The third will find files1.txt but not files.txt.

It is helpful to use ls to list what you are planning to remove, to make sure you don't accidentally delete the wrong thing!

Some other useful (or fun) commands are:
top - see who's doing what on the system (q to quit). You can look at a specific users processes on our machines by typing u, and then typing the username and hitting enter
info - like man, but more comprehensive (q to quit)
date - get the date
head - get the top few lines of a file
tail - get the last few lines of a file

Programs, Commands, and $PATH

Less and nano are all programs, but ls, man, cp, etc are commands. What is the difference? Commands are built into the shell, and some programs are as well. However, where they exist is slightly different and very important.

A program is an executable file, that the operating system can read and run. We've run echo, but what IS echo?

cd /bin
less echo

If we try to look at the file, it tells us it's in binary. We've looked at it anyway and its pretty much gibberish. This is because it has been translated into something that the machine can read

(but we can't).

Sometimes we'll want to run programs with their full path name (from root) to make sure that our shell knows where to find it.

/bin/echo hello class!

How come we don't have to always define the full path if the file isn't in our folder? This phenomenon is the result of a very important shell variable $PATH.

echo $PATH

As you can see, /bin is in there. Also note that . is in PATH as well, which makes it so we don't technically need to call any program we write ./program.ba, we could just use program.ba. However, it is generally safer to make sure you are calling the right program (who knows what is in all those folders!). All the folders are separated by a :. It is important to note that bash takes the FIRST instance of a program it comes across in the order of the folders in the PATH variable. If you have two copies of a program, it will run only the first it comes across. This is again why ./ is helpful, since it is usually one of the last folders listed.

Because of this order rigmarole, sometimes we want to know where a program is, to make sure it's the correct one.

which less

This tells you from where less is being executed. Let's try looking for cd. If we call:

which cd

though... it says there is no cd in our path anywhere. That's because cd isn't a program, it's a command built into bash. Doesn't change anything, just interesting to note.

If I wanted a program I wrote to run anywhere I am, I would have to add the folder it lives in to my PATH variable, or more conveniently, put it in a folder that is in my path and holds all my programs. This is the /bin. We don't have permission to add things to /bin, so we will make our own.

cd $HOME
mkdir local
cd local

mkdir bin

Local/bin is a very common folder for installations, and some programs will look for it when installing. Now let's add it to the path, so Unix can find programs we write or install.

declare PATH=$PATH:<NEW DIRECTORY>

Remember the declare commands we carried out earlier? By saying PATH=$PATH:newdirectory, we're appending the new stuff to what is already stored in PATH. Remember that : separates directories in your path. You can add several at a time if you'd like:

declare PATH=$PATH:<NEW DIRECTORY>:<NEW DIRECTORY>: (etc)

However, if we sign out, this variable is lost. Variable definitions are cleared every time you leave bash. How do we add these permanently? By using that .bashrc file I mentioned the other day. It is an administrative file, with our profile information in it.

Modifying .bashrc

First, let's get to the file:

cd ~
nano .bashrc

Find the line that says 'export PATH=' – if it doesn't exist, add it! Export is another way to declare variables in the shell (see the discussion board for more). Add a colon after the last directory listed, and insert the ABSOLUTE path to your bin directory. Save and exit.

You have to restart bash to affect change (it's read only at log in usually). You can either exit out to csh and back in, or you can type:

source .bashrc

This command tells the shell to run the current .bashrc without having to log out. Now let's check to see if it worked:

echo $PATH

******Make it your own******

Here are a couple more changes to .bashrc to make your life better.

1) Open .bashrc again, and find the PS1 variable, which determines the text for the prompt (if it's not there add a line that says PS1=)There are a few special instructions you can give the prompt to display variables:

\u = user\h = hostname\w = current working directory

You can replace \w with $(pwd) to give you the entire path instead of just the current directory (which is how I have it sometimes).

declare PS1="\u@\h \W$ "

But... that's boring. Here's how to make a really snazzy prompt:

http://bashrcgenerator.com/

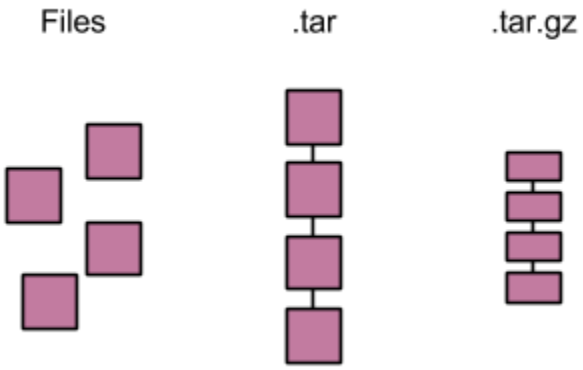
Make it however you want, in a point and click manner, and then paste that PS1 variable command that it gives you into your .bashrc file. Save, exit, and source.

2) Adding in some aliases:Find where aliases are in the .bashrc file, so we can add below them to keep our file organized. To make your own, add a line like: "alias ls='ls –-color' ". Another is to make nano start without word wrapping. Word wrapping is nice for human eyes, but can mess up your coding. "alias nano='nano -w' ".

Tarballs and Gzip

Often files are compressed to make it easier to transfer them. Compression is very common with Illumina files and often with program downloads. In most familiar systems like Windows, these are .zip files. In Unix, these are gzipped, or .gz files. You may see them as .tar files as well, or often, .tar.gz.

Tarring links files together into a group, like a folder that is a single object to easily move. They are stuck together with tar (I warned you there are puns everywhere). Gzipping compresses the files, making the tarball (real term) easier to push around. These two things are very often used in conjunction.



So how do we untar and ungzip?

Because the file extension on the end is gzip, we have to undo that first. We will use the program gzip.

man gzip

It looks like we will need the -d flag.

gzip -d file.tar.gz

NOTE: This program will not run on files without the .gz extension.

mv file.tar.gz file.tar.haha
gzip -d file.tar.haha
gzip: file.tar.haha: unknown suffix -- ignored

Once we have decompressed the file, we are left with a tarball. Notice it is a bit bigger:

ls -l

We can tar or untar with the program tar.

man tar

That third example sounds like what we need. We will need the x and f flag (x is for extract, and f tells it that you are taking it from the file you are giving it).

tar -xf file.tar.gz

You can actually get tar to ungzip for you as well, since these two processes are so commonly used together

tar -zxf file.tar.gz #z for gzip

There are two other common compressions, bzip2 (.bz2) and zip (.zip), which we won't go through, but are on the cheat sheet. Additionally, if you ever see a file extension that you don't know, google it!

You can also tar and gzip a file as follows (note that the default is to create the compressed tarballs, not decompress them; hence the flags):

tar file.txt
gzip file.txt

NOTE: The defaults for both tar and gzip are to CREATE compressed folders, yet the most common usage (and therefore what we just went over) is to use these commands to unzip and untar files. Why is the default to create and compress? BECAUSE BACKING UP FILES IS IMPORTANT AND SPACE IS LIMITED. Remember how we said rm is forever? It's generally a good idea to have a backups folder of anything important. As space is limited, it's also important to use space frugally - hence tar and zip.

tl;dr - USE THESE COMMANDS TO BACKUP COMPRESSED VERSIONS OF YOUR DATA ^_~

******NewData.tar.gz******

To check disk space before you start moving files into your directory, you can use the two different commands:

du -hcs #for use in Unix

I showed you Filezilla and scp earlier, but there is another command when you are getting files from the internet, which is common for source code. It eliminates the requirement of downloading the files to your machine, and instead download them directly to the cluster space:

wget <url>

We can go to the course website, copy the URL and wget the data right to our folder.

```
wget http://darlinglab.org/data/demux/Undetermined_S0_L001_R1_001.fastq.gz
gzip –d Undetermined_S0_L001_R1_001.fastq.gz
wget http://darlinglab.org/data/demux/Undetermined_S0_L001_I1_001.fastq.gz
gzip –d Undetermined_S0_L001_I1_001.fastq.gz
```

Let's practice a couple commands first. Start by tarring files together:

```
tar -cf NewData.tar Undetermined_S0_L001_R1_001.fastq
Undetermined_S0_L001_I1_001.fastq
gzip NewData.tar
```

And now practice unzipping and untarring at the same time:

```
tar -zxf NewData.tar.gz
```

Or

```
gzip -d NewData.tar.gz
tar -zx NewData.tar
```

...and look at the Illumina file:

```
less Undetermined_S0_L001_R1_001.fastq
```

Fasta/Fastq Format

Being able to generally understand a flat file is useful, so that you can generally scan though the file and see the quality, know when things are being cropped incorrectly, and just for curiosity ^_^.

Fasta files have two line types - sample headings (designated by a >) and sequences (lack a > designation):

Fasta Format:

```
>SAMPLE_NAME
gtcagatctcagtagc #sequence
>SAMPLE_NAME
gtcagatctcagtagc #sequence
```

Because the headings are designated by a special character, sequences can be on multiple lines without confusing the machine or ourselves. Anything after a heading that doesn't begin with a > is assumed to be part of the same sample:

>SAMPLE_NAME
gtcagatctcagtagc #sequence
gtcagatctcagtagc #sequence line 2
gtcagatctcagtagc #sequence line 3
gtcagatctcagtagc #sequence line 4

>SAMPLE_NAME
gtcagatctcagtagc #sequence
gtcagatctcagtagc #sequence line 2

>SAMPLE_NAME
gtcagatctcagtagc #sequence

Fastq files have a bit more detail to them, particularly when they come from an illumina machine. The first line is a header (designated by an @), the next line is the sequence (similar to fasta), then there is a comments line (+), and a base call quality line (of the same length of the sequence line).

Illumina Raw Data - Fastq format (see PPT for more information):

For example:

```
@FCC1PFHACXX:2:2316:19544:100787#ATCACGA/1
CGGTCCAACAATTCTGTGCAAAAAAAAAAATTTTCAAGAGGGGAGGGGGGGGAGAGATGGCGG
+\^^\``^acYbcchhddBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
```

Each part of this has information coded into it. In parentheses, I have labeled each of these parts with the information they refer to:

```
@ (start) FCC1PFHACXX (machine name):2(flow
cell):2316(individual tile):19544(x coordination):100787 (t
coordinate) #ATCACGA(tag)/1(pair number)
CGGTCCAACAATTCTGTGCAAAAAAAAAAATTTTCAAGAGGGGAGGGGGGGGAGAGATGGCGG
GGCCTAGGCACGAACAAAACACCAAGCCAGCGAACA(seq)
+ (optional info - missing in this case)
```

```
\^^``^acYbcchhddBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BB BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB (quality score for each
position in the sequence - required to be the same length).
```

Quality Scores (see PPT for more information)

One of the first things you will notice in these quality scores is a lot of "B"'s everywhere. These are related to the quality score, and are in what is called phred format, which was developed for the human genome project. It is based on the original Sanger style sequencing, where chromatograph peak shapes were used to determine how accurate the read for any one base pair was (picture). This information was encoded in a log scale from 1 (terrible) - 50 (the best - approximately a 1 in 100000 chance it's an error).

This scale is all well and great, until you want the same number of characters for your quality scores as you do for your sequence - meaning the two digit phred score has to be translated into a one character code. If quality control scripts or whatever you are doing results in different a different number of characters in the sequence than in the quality scores, upstream programs will not be able to read this information (important to consider when writing your own scripts and trouble shooting in general).

This problem is solved by using something called ASCII codes, which are numbers that map to a single characters. However, to avoid weird characters in the lower range of ASCII codes that might not do good things for text based files, 65 is added to the phred score (65 is 'A' in ASCII). This means all those 'B's are a phred score of 1 (66 - 65), or are complete junk, as is basically anything that is a capital (phred score of <26). This is good to know, since you should see substantially less 'B's and capital letters in your quality controlled data as compared to your raw reads - a quick check for quality control scripts!

There are different versions of quality score are as follows:

S - Sanger Phred+33, raw reads typically (0, 40)
X - Solexa Solexa+64, raw reads typically (-5, 40)
I - Illumina 1.3+ Phred+64, raw reads typically (0, 40)
J - Illumina 1.5+ Phred+64, raw reads typically (3, 40)
L - Illumina 1.8+ Phred+33, raw reads typically (0, 41)

This is generally not a huge problem, with the exception of Illumina 1.3 and Illumina 1.5 scores... because they do not appear any different, they are just coded differently. I will show you

how to tell, because some programs (such as the Genomics Analysis Toolkit, vastly used in RAD-tag analysis and RNA-seq) crash if you do not adjust the scores. Determining your formatting is difficult because Illumina started as 1.3, then went to 1.5, then went BACK to 1.3. Just keep this in mind - if a program complains that the quality scores are weird and crashes, it means you will have to convert the scoring (with your new magical bioinformatic skillz) or tell the program to read it correctly (more common).

****Fastqc *(see PPT for more information)*****

less Undetermined_S0_L001_R1_001.fastq

Not a lot of capitals or Bs! Good news!! But we don't want to scroll through all of this and we don't know what format it is in.

There is a wonderful program called Fastqc (another pun). It is easy to call:

fastqc Undetermined_S0_L001_R1_001.fastq

However, his program requires java to run.Many programs we have installed on the cluster are in modules. In order to load a module on the system, in this case fastqc:

module load fastqc

Oh no, it complains that it needs java! So we have to load that first. But… there are more than one copy of java on our system! How do we find modules?

module av #for available; lists all modules available on the system
or
module av java #lists only modules with the name java

There it is! Let's load java/1.7.0_40

module load java/1.7.0_40

and then fastqc!

module load fastqc

No problems that time! Now we can call fastqc as normal:

fastqc Undetermined_S0_L001_R1_001.fastq

NOTE: If you are curious if a module will need something to be loaded first, use "module display <name>" and it will tell you about the module, purpose of the program, and any dependencies it has!

The output is a html document (ls to see), so it will be easier to just download it to your computer with filezilla and open.

NOTE: This is a great video that talks about all the information in the FastQC output: https://www.youtube.com/watch?v=bz93ReOv87Y

Fantastic. We now know our data is in Illumina 1.9 coding, is high quality, and seems to be reasonable. You have completed the first step that you will likely do in any initial bioinformatics!

PPT that accompanies above notes

Fasta Format

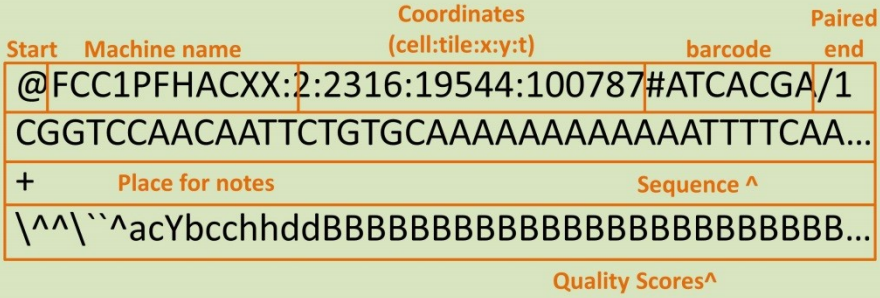
```
>Sample_name|information|information|etc.
CGGTCCAACAATTCTGTGCAAAAAAAAAAATTTTCAA
GAGGGGAGGGGGGGGAGAGATGGCGGGGCCTAGGC
ACGAACAAAACACCAAGCCAGCGAACA

>Sample2_name|information|information|etc.
CGGTCCAACAATTCTGTGCAAAAAAAAAAATTTTCAA
GAGGGGAGGGGGGGGAGAGATGGCGGGGCCTAGGC
ACGAACAAAACACCAAGCCAGCGAACA
```

Illumina Raw Data (Fastq)

@FCC1PFHACXX:2:2316:19544:100787#ATCACGA/1
CGGTCCAACAATTCTGTGCAAAAAAAAAAATTTTCAA…
+
\^^\``^acYbcchhddBBBBBBBBBBBBBBBBBBBBB…

Illumina Raw Data (Fastq)



What's with all the B's?

Phred quality scores are **logarithmically** linked to error probabilities		
Phred Quality Score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90%
20	1 in 100	99%
30	1 in 1000	99.9%
40	1 in 10000	99.99%
50	1 in 100000	99.999%

We want 99% accuracy or better, meaning scores of 20+. However, if we put these in the same order as the sequence, the spacing gets offset quickly and much harder to parse:

CGGTCCAACAATTCTGTGCAAAAAAAAAAAATTTTCAA

20**30**22**9**....

What's with all the B's?

Phred quality scores are **logarithmically** linked to error probabilities		
Phred Quality Score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90%
20	1 in 100	99%
30	1 in 1000	99.9%
40	1 in 10000	99.99%
50	1 in 100000	99.999%

To avoid this offset, these scores are coded as single letters, using an old numbering system in computer science (ASCII). All characters have a number associated with them, with A=65 (a bunch of punctuation starts before letters).

ASCII Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

What's with all the B's?

Phred quality scores are **logarithmically** linked to error probabilities		
Phred Quality Score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90%
20	1 in 100	99%
30	1 in 1000	99.9%
40	1 in 10000	99.99%
50	1 in 100000	99.999%

To avoid this offset, these scores are coded as single letters, using an old numbering system in computer science (ASCII). All characters have a number associated with them, with A=65 (a bunch of punctuation starts before letters).

We want our scores to start with A because a bunch of punctuation is coded 0-64, and we don't want to confuse the computer with special characters.

Letter code = 65 (starts at A) + Phred Quality Score

What's with all the B's?

Phred quality scores are **logarithmically** linked to error probabilities		
Phred Quality Score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90%
20	1 in 100	99%
30	1 in 1000	99.9%
40	1 in 10000	99.99%
50	1 in 100000	99.999%

For example: A quality score of 20 is coded as:

Letter code = 65 + 20 = 85

Letter code =U

Inversely:

Letter code = B (66) = 65 + Score

Score = 1

Basically, all those B's are very poor quality sites (1 in 1 probability of incorrect)…

The scale is subject to change…

To make matters worse, the scaling has changed slightly through time. Illumina has at least two different coding systems which have been used intermittently across different years.

Why do we care?

You might not, but the analysis programs sure do. They will error out if they are getting letter codes outside of their range. Knowing what this means makes fixing the issue much easier.

Cluster Quick Guide: Computers you can log into

karst.uits.iu.edu

1 node, 16 cores, 16GB ram per core, for general jobs

mason.indiana.edu

18 nodes, 32 cores per node, 512GB ram per node, for high memory genomics jobs.

How much space do I have?

You can check it with:

quota

Modules

Modules make certain programs available to you in your path without having to find/install them.

View module list:

module av

Load a module (for example "bio" which contains useful bioinformatics programs like trinity):

module load <module name>

Unload a module (if you for some reason need to)

module unload <module name>

List the modules you have loaded

module list

Determine if a module needs another module loaded first (dependency) and to get more information on the module in general.

module display <module name>

Job Queues

You can run jobs without having to be signed in or wait on them. They are submitted to the

"queue". Any job that will take more than 20 minutes must be submitted to the queue (there is one for Mason and one for Karst, just depends on where you are logged in).

Job Submission

In order to submit to the que, you need to make a job file. They work kind of like an executable.ba would - they run the code in the file. However, additional information is needed in the header. It looks like this (all #comments are added):

```
#PBS -k oe #
#PBS -m abe #mail me when job : a – abort, b - begins, e - ends
#PBS -M <your email>
#PBS -N <name of job>
#PBS -l nodes=1:ppn=1,vmem=16gb,walltime=2:00:00 #See note below

#The –l flag requires you to set how long and what resources you are requesting. Please see https://kb.iu.edu/d/bdkd for information on this setting.

#Load modules required
module load java/1.7.0_40
module load fastqc

#you must enter the directory where your data is, with an absolute path
cd /N/dc2/scratch/ss93/Analysis

#call the program as you would normally
fastqc NewData.fq
```

Managing your jobs

What is the status of my job (for example if you are me/ss93)?

```
qstat -u ss93
```

NOTE: q - waiting in the que, r – running, c - complete

Delete a job (the number is listed when you use qstat

```
qdel -j 60123
```

What's going on with my job?

Your job will create a file, entitled whatever you used in the -N flag above (i.e.

RunTrimmerMagna1) followed by a .o and the job number (i.e. RunTrimmerMagna1.o23453). You will also see a .e file of the same sort (i.e. RunTrimmerMagna1.e23453). These are your STDOUT and STDERR outputs, respectively. They will give you the output on the job

Tips

- Try running commands before putting them in a job file. This ensure they will run correctly. You can always stop them with ctrl c.

- Create a blank job file and just copy it to another file name when you are creating a new job submission

- Have some sort of naming paradigm for jobs. I call all mine RunWhatever. Some people give them a .job extension. It doesn't matter what you do, but it helps when you are looking for them in your folders.

- You can run multiple things in parallel if you put an '&' at the end of each command and then 'wait' at the end of the file. This will run all commands in the background at the same time, and wait until all background jobs are done before terminating. If you forget 'wait' the job will launch, push everything to the background, run out of code to run, and terminate. If you use &, you must use 'wait'!

For more in depth information, try the KnowledgeBase:

https://kb.iu.edu/index.html