Module 1

Part I: Essential Unix/Linux

Goal: To gain familiarity with, and comfort using, the Unix command line. The majority of bioinformatics programs are command line tools. Competency with the command line will make navigating these tools and interpreting/re-organizing/filtering their outputs much easier.

1.1 Getting connected and reconnected

The steps for connecting to your virtual machine (VM) will differ depending on what operating system you are running. Follow the steps in the subsection below that corresponds to your system.

On a PC

The instructions below describe two methods of connecting to your VM from a PC—**OpenSSH/PowerShell** and **PuTTY**. You only need to follow the instructions for one method. **OpenSSH** requires no additional installation on computers running the latest version of Windows. **PuTTY** requires installation but may be more user-friendly.

OpenSSH/PowerShell

Open the **PowerShell** program. You can use the OpenSSH client **ssh** (secure shell) to connect to your virtual machine:

- ☐ Let's try and connect. Type the command below into **PowerShell**, changing myName and Xs to match the credentials you received via email. myName should match the username you received, and Xs should be changed so that the hostname below (the part after the "@" symbol) matches the hostname you received. When you have finished typing the command, hit enter to run the command.

```
•  ssh myName@XX.XXX.XXX.XX
```

- ☐ You will be prompted for a password; type the password we sent you in the email. When you start typing your password, you won't see anything appear, but **ssh** is nevertheless reading the characters you type. Hit enter when you are done.
- ☐ After successfully logging in, you should now see a prompt, similar to:

```
myName@ip-XXX-XX-XX-XXX:~$
```

You will see your name instead of **myName**, and you will see new numbers in place of the Xs. You are now "located" in your home directory (often abbreviated as ~ or ~/). The dollar sign $ at the end of prompt simply signifies the end of the prompt.

- ☐ To logout from your VM, just type:

```
• logout
```

This will close the connection, returning you to the **PowerShell** prompt on your PC.

- ☐ Try and log in again. We will assume that you can do this step in future classes without issue.

PuTTY

PuTTY is a free telnet/SSH client for Windows that predates the native Windows OpenSSH client. Although it is not necessary to install a separate SSH client for this workshop, you may prefer **PuTTY** for its user-friendly interface and better terminal emulation. You can download **PuTTY** from https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

- ☐ Download and install **PuTTY** from the link above.
- ☐ Open **PuTTY**. (If you ran the installer, you can find **PuTTY** in the Start menu.)

  Enter the hostname for your virtual machine into the "Host Name (or IP address)" box. You can find your hostname in the email we sent you with your VM credentials; it should look like `myName@XX.XXX.XXX.XX` where each set of Xs represents between one and three digits. Leave the default port 22 unchanged.

- ☐ Click "Open" to connect to your VM.
- ☐ You will first be prompted for your username. Enter the username from the credentials email. When you start typing your password, you won't see anything appear, but **PuTTY** is nevertheless reading the characters you type. Hit enter when you are done.
- ☐ When prompted for a password, enter the password from the credentials email.
- ☐ After successfully logging in, you should now see a prompt, similar to:

```
myName@ip-XXX-XX-XX-XXX:~$
```

You will see your name instead of **myName**, and you will see new numbers in place of the Xs. You are now "located" in your home directory (often abbreviated as ~ or ~/). The dollar sign $ at the end of prompt simply signifies the end of the prompt.

- ☐ To exit **PuTTY** just type exit:

```
• exit
```

This will close **PuTTY**.

- ☐ Try to start **PuTTY** and log in again. We will assume that you can do this step in future classes without issue.

1.1.2 On a Mac

If you are working on a Mac, simply open the program **Terminal**, found in the *Utilities* folder. You can use the **ssh** (secure shell) command to connect to your virtual machine.

- ☐ Let's try and connect. Type the command below into **PowerShell**, changing myName and Xs to match the credentials you received via email. myName should match the username you received, and Xs should be changed so that the hostname below (the part after the "@" symbol) matches the hostname you received. When you have finished typing the command, hit enter to run the command.

```
• ssh myName@XX.XXX.XXX.XX
```

- ☐ You will be prompted for a password; type the password we sent you in the email. When you start typing your password, you won't see anything appear, but **ssh** is nevertheless reading the characters you type. Hit enter when you are done.
- ☐ After successfully logging in, you should now see a prompt, similar to:

```
myName@ip-XXX-XX-XX-XXX:~$
```

You will see your name instead of **myName**, and you will see new digits in place of the Xs. You are now "located" in your home directory (often abbreviated as ~ or ~/). The dollar sign $ at the end of prompt simply signifies the end of the prompt.

- ☐ To logout from your VM, just type:

```
• logout
```

This will close the connection, returning you to the prompt on your Mac.

- ☐ Try and log in again. We will assume that you can do this step in future classes without issue.

1.2 Exploring and learning basic commands

Let's try to familiarize ourselves with our VM by using a few commands.

- ☐ Display your working directory using **pwd** (print working directory).

```
• pwd
```

You will see something like this:

```
myName@ip-xxx-xx-xx-xxx:~$ /home/myName
```

This is your current working directory, meaning you are virtually located here. Any actions you perform will assume you are here and will create files here unless you instruct otherwise.

- ☐ Let's list the current directory to see what goodies we can find here. The required command is **ls**:

```
• ls
```

```
myName@ip-xxx-xx-xx-xxx:~$ ls
assembly  blast  genes  qiime2  rnaseq  sequences  variants
```

- ☐ Let's create a *unix* directory. This is where we will do all our work on our Unix skills.

```
• mkdir unix
```

- ☐ List the current directory again to make sure the *unix* directory was successfully created:

```
myName@ip-xxx-xx-xx-xxx:~$ ls
assembly  blast  genes  qiime2  rnaseq  sequences  unix  variants
```

How do we know it's a directory? One way is to tell is that the name is colored blue—more on this later...

- ☐ Change into the new *unix* directory.

```
• cd unix
```

```
myName@ip-xxx-xx-xx-xxx:~/unix$
```

- ☐ To illustrate how to navigate the UNIX directory structure, we'll create a new directory inside the current one:

```
• mkdir example
```

This has created a new directory called *example* inside the *unix* directory.

- ☐ List your current directory's contents again. You will now see a new directory called *example* has been added:

```
myName@ip-xxx-xx-xx-xxx:~/unix$ ls
example
```

- ☐ Now, we'll create an empty file using the **touch** command:

```
• touch myFile.txt
```

☐ List your current directory's contents again. Now we have *myFile.txt*:

```
myName@ip-xxx-xx-xx-xxx:~/unix$ ls
      example  myFile.txt
```

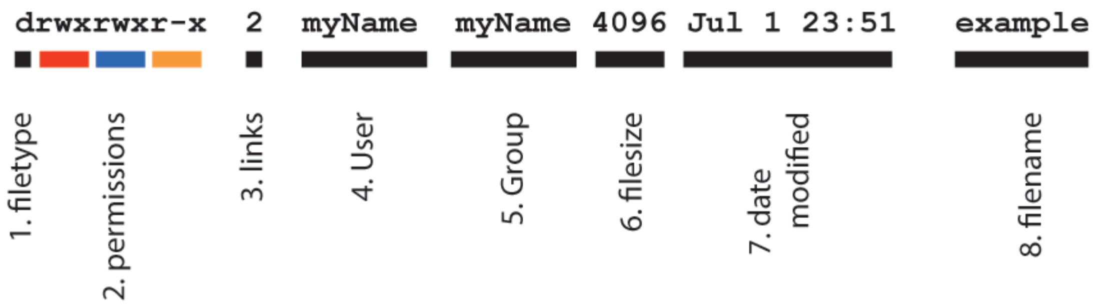
Now we see that files are listed in black. Another way to tell whether you have a directory or a file is to use **ls** with the -l option (lowercase letter L—not the number one). This is the "**l**ong" directory listing that shows a lot more details:

☐ Perform a long directory listing.

```
• ls -l
```

```
myName@ip-xxx-xx-xx-xxx:~/unix$ ls -l
total 4
drwxrwxr-x 2 ngsadmin ngsadmin 4096 Jul 26 22:06 example
-rw-rw-r-- 1 ngsadmin ngsadmin    0 Jul 26 22:06 myFile.txt
```

What does this all mean? Below is an example key:



For file permissions, "r" means "can read", "w" means "can write", and "x" means "can execute." Notice there are nine columns (or three sets) of permissions in the example; these three sets of permissions designate the settings for the user/owner (in red on the key), the group (blue), and others (everyone else on the machine, in orange on the key).

So, other than by simply remembering it, how do you know what "-l" does?

☐ Show the manual for **ls**:

```
• man ls
```

Every Linux program has a manual page installed and is accessible via **man *program-name***.

☐ Use the spacebar to scroll through the document and find the -l option

☐ When you have finished exploring other possible options, press **q** to quit the **man** program.

1.3 "There's no place like home"

As a beginner, it's easy to get "lost" in the UNIX directory structure until you learn the concept of directory/file paths. However, a quick and easy way to "find" yourself is to return to your home directory. The following is an exercise to illustrate absolute and relative paths and to show you how to get home from any location.

☐ Let's begin learning about directory "paths." First, we need to know where we are in the system. Recall, we can find this out with **pwd** (print working directory).

```
• pwd
```

Unless you did something unexpected, you should still be in the *unix* directory. **pwd** prints out an **absolute path**:

```
myName@ip-xxx-xx-xx-xxx:~$ /home/myName/unix
```

An absolute path to a file or directory points to a specific location and is unaffected by where you are currently working in the filesystem.

/	The root directory that houses the directories that contain all the system resources
home	where all individual User home directories are located
myName	your personal home directory
unix	the directory in which you are currently working (and where all output files will be created unless you specify otherwise

☐ If we want to find our way home, we can get there in a few different ways. First, let's use an absolute path. (Note: change myName to your username on your VM.)

```
• cd /home/myName
```

This says start at the *root* directory, move into *home* and then into the directory called *myName*. You can tell if this worked because the command line prompt should now say (remember ~ means your personal home directory):

```
myName@ip-xxx-xx-xx-xxx:~$
```

☐ Now we're going to jump all the way to the *example* directory that we created a few minutes ago—this time using a relative path.

```
• cd unix/example
```

This path is a **relative path** because the meaning of the path depends on the current working directory. The command above says change into the *unix* directory, found inside the current directory, and then change into the *example* directory, which is inside in the *unix* directory.

```
myName@ip-xxx-xx-xx-xxx:~/unix/example$
```

☐ If we want to make absolutely sure of our location, we can print our working directory (**pwd**).

```
• pwd
```

We should get the absolute path to the *example* directory:

```
/home/myName/unix/example
```

☐ What if we try to run the previous command again:

```
• cd unix/example
```

```
-bash: cd: unix/example/: No such file or directory
```

This happens because the command says change into the *unix* directory inside the current one but there is no *unix* directory here. How do we know that? We can list the current directory.

☐ Now, we'll use a relative path to start navigating back to our home directory:

```
• cd ..
```

This says move up one directory level which, in this case, places us in the *unix* directory:

```
myName@ip-xxx-xx-xx-xxx:~/unix$
```

☐ Let's repeat the previous command to move up one more level, which places us back in our home directory.

```
• cd ..
```

```
myName@ip-xxx-xx-xx-xxx:~$
```

☐ To help us understand the difference between true /*home* and <u>your</u> home directory, let's list the root-level /*home* directory.

```
• ls /home
```

This will show a listing of all the "home" directories for every user on the system. Look for your own username; it will surely be there. The forward slash in front of *home* simply tells the shell to look for the home directory found in the systems root directory.

☐ Now list your personal home directory. (Remember that tilde represents <u>your</u> home directory.)

```
• ls ~
```

This shows the directories and files that are available for your personal use.

☐ Now, we'll use the UNIX equivalent of clicking our heels to go straight to our personal home directory:

```
• cd
```

We can also specify home using the ~ (tilde) symbol (`cd ~`) for changing directories, but that's just too much unnecessary work!

```
myName@ip-xxx-xx-xx-xxx:~$
```

1.4 Improving productivity - using your command history:

Bioinformatics involves many repetitive tasks. The UNIX system has a number of useful features to assist us. First, it keeps track of all the commands that you have previously issued up to a specified limit. You can iterate back and forth through previous commands by using the ▲ and ▼ keys on your keyboard.

☐ Try it:

- ▲ ▲ ▲ ▲ ▼ ▼ ▼ ▼

☐ Use the up and down arrow keys to navigate to the previous command where you "jumped" to the *example* directory inside the *unix* directory (cd unix/example). When you find it, hit return.

```
myName@ip-xxx-xx-xx-xxx:~/unix/example$
```

☐ You can also see your command history as follows:

```
•  history
```

In the last two exercises, we had you move between different directories to teach you about "paths." However, once you settle on a working directory for a given project, there is no need to change directories again. In fact, until you are familiar with the command line environment, you will get into trouble if you change directories unnecessarily. For this reason, we will learn how to work inside one directory and operate on other directories and files by specifying the "paths" to these resources.

BUT FIRST…

1.5 Minimizing errors with tab completion

We cannot overemphasize the importance and utility of learning and using tab completion. Ninety percent or more of command failures can be eliminated simply by using tab completion routinely.

☐ **Make sure you are in your personal home directory**.

☐ Create a file called *testfile.txt* in the *example* directory you created earlier:

```
•  touch unix/example/testfile.txt
```

☐ Type `ls u`, then hit the tab key but DON'T hit return:

```
•  ls u<tab-key>
```

The cursor will complete the second word to "unix" because it knows there is only one directory inside your current working directory that starts with the letter "u."

- ☐ Now add an "e" after the forward slash and hit tab again:

```
•  ls unix/e<tab-key>
```

- ☐ Now you can finally hit the return key, and you will get a file listing for the *example* directory that is present in the *unix* directory. It should contain the *testfile.txt* file.

```
ngsadmin@ngs-02:~$ ls unix/example/
testfile.txt
```

What if you had two directories that start with the same letter? To demonstrate, we'll create a second directory that starts with "ex" inside the *unix* directory. Remember, however, we are currently in the *example* directory, so we need to make sure to tell the shell to create our new directory in *unix*.

- ☐ Create a directory called *extra* inside *unix*. First, we'll use an explicit path.

```
•  mkdir ~/unix/extra
```

Here is an example of where ~ (tilde) comes in useful. Instead of typing the full explicit path, which would be */home/myName/unix/extra*, we can use ~ which is shorthand for */home/myName*.

- ☐ Now make a second directory in the *unix* directory using a relative path. This would also be a good time to try using the up-arrow key to repeat the previous command after removing the "~/" and adding a "2" the end of "extra".

```
•  mkdir unix/extra2
```

Here you're telling the shell to create the *extra2* directory inside the *unix* directory which can be found in your current working directory.

- ☐ Let's make sure that the proper directories were created:

```
•  ls -l unix
```

```
myName@ip-xxx-xx-xx-xxx:~$ ls -l unix
total 12
drwxrwxr-x 2 ngsadmin ngsadmin 4096 Jul 18 20:55 example
drwxrwxr-x 2 ngsadmin ngsadmin 4096 Jul 22 06:20 extra
drwxrwxr-x 2 ngsadmin ngsadmin 4096 Jul 22 06:20 extra2
-rw-rw-r-- 1 ngsadmin ngsadmin    0 Jul 26 22:06 myFile.txt
```

- ☐ Now try and use tab completion to list the *unix/example* directory. You will find that u <tab> will complete as expected, but e <tab> will only complete as far as "ex". This because the system now sees three directories that start with "ex".
- ☐ Hit tab twice to see the possibilities available for completion.

```
example/  extra/  extra2/
```

Of course, we knew what to expect because we just created these directories, but, in many cases, we will not know a directory's contents, or we will have forgotten. Double tab comes in very useful in these situations.

- ☐ Type an "a" and then tab, and the command should now be able to complete to *example*. You can hit return to execute the command.
- ☐ To illustrate the error-checking capabilities of tab completion, try typing **ls unix/E** and hitting tab:

```
•  ls unix/E <tab-key>
```

Completion will fail and your machine may even bloop at you to say something's wrong. This illustrates three things:

i) Unix is case sensitive, and, in this case, it does not see any directories starting with capital "E" inside the directory that is one level up from your current location.

ii) If tab completion is not completing, this means YOU made an error. Either you are trying to point to a directory/file that does not exist in the specified location, OR you made a typo when typing its name. (Here we made a typo— "E" instead of "e".)

iii) If you use tab completion, it is impossible to make typos or to run programs using files that are not in the locations where you think they are.

For the best experience in this workshop, and in the future, you should get in the habit of using tab completion right now and keep using it routinely for all commands.

1.6 Learning a text editor

Most of the text editors with which we are familiar (Word, GoogleDocs, TextEdit, etc.) use symbols for hyphens, quotation marks, and end-of-line markers that are incompatible with many unix programs and can create problems that are VERY hard to diagnose. For this reason, it is useful to learn how to use a UNIX text editor. Most programmers will use sophisticated editors such as **vi**, **vim**, **emacs** or **joe**, but these have steep learning curves and can be confusing for beginners. For this reason, we will learn **nano**, which, as its name implies, is a program with a small but decent set of "bare-bones" functions.

- ☐ **Make sure you are in the *unix* directory**. **We are not going to move out of this directory for the rest of this exercise.**
- ☐ Start **nano**, and we will edit a file called *name.txt*. **nano** will create the file if it does not exist. You can add a directory path before the filename to specify where it should go. (By default, the file would be created in your current working directory.) The following creates and edits a file called *name.txt* within the *example* directory:

```
•  nano example/name.txt
```

Remember: we don't need to be in the *example* directory to create a file there. We simply tell the system where we want the file to be created by specifying the path.

- ☐ The **nano** screen has two main sections: a buffer to edit text and a menu of common shortcuts. The ^ character in the shortcuts section represents the control key. For example, ^O means hold the control key and "o" key together. Shortcuts in **nano** do not use the shift key, so ^O really means ^o. WriteOut means save. (Think of it as writing out to disk.)
- ☐ Test **nano** by entering your name.

- ☐ Save your file. (^o).
- ☐ **nano** will prompt you to give a filename for the file you will save. Hit enter to write to that file. By default, this is the file you gave to **nano** (*example/name.txt*) when you first issued the command. You can also start **nano** without specifying a filename.
- ☐ Exit **nano** (^x).
- ☐ Use **nano** to open your file again and verify its contents to convince yourself that the text was saved. **Remember to specify the file path (*example/name.txt*).**

Pressing ^g (the control key and g together) will display common **nano** shortcuts. You can quit the shortcuts menu by hitting "q". ^c will display a status bar with your current position (line, column, and character numbers).

Copying and pasting can be difficult at first. When using **PuTTY**, highlighting with the mouse selects and copies text; right-clicking pastes text. In **PowerShell**, you can highlight text with your mouse and press the enter key to copy text, and, as in **PuTTY**, you can right click to paste text. With the Mac, you can copy and paste as usual.

Within **nano**, ^k will cut the current line and ^u will paste. You can also copy the current line by pressing the shift key, alt key, and 6 key simultaneously. There might be a noticeable pause when pasting large amounts of data.

You can use **nano** to view a text file safely; if you accidentally edit the file, just exit without saving by answering "no" when prompted.

If you only need to view a document without editing it, you can use the command **cat** (concatenate), for short files.

```
cat example/name.txt
```

Or, for long files, **more** and **less** are commands that will let you page-up and page-down through the file. They are often more useful than **cat** for viewing large files.

```
less example/name.txt
more example/name.txt
```

You can exit from **more** and **less** using **'q'** (and with ctrl-c, for **more**).

1.7 Using SCP to transfer files between machines

scp (secure copy) is a command-line tool that can send files to a remote machine or download files from a remote machine. Let's learn how to use **scp** by sending your newly created *name.txt* document to an Administrator Virtual machine.

SCPing to a remote machine involves the following syntax: 1) first specify the file (path-to-file/filename) to be transferred; 2) define the destination and new name (if needed) of the copied file (machine-address:path-to-file/filename):

```
scp <file_to_transfer> username@remote.machine.address:target_directory/filename
```

File transfer will begin upon entry of a valid password.

- ☐ Make sure you are still in the *unix* directory:

```
myName@ip-xxx-xx-xx-xxx:~/unix$
```

- ☐ Not only will we send the file to the Administrator machine, but we will change its name in the process. If we don't do this, each time a class participant sends their file to the Admin VM, the new file will simply write over any similarly-named versions uploaded beforehand. Type the following, but replace `myName` with the name you gave in the *name.txt* file that you created using **nano**.

```
scp example/name.txt ngs@35.171.154.104:Desktop/myName.txt
```

- ☐ The first time you connect, you will probably get a message similar to the following:

```
The authenticity of host ' myName@ip-xxx-xx-xx-xxx (xx.xxx.xxx.xxx)' can't be
established.
ECDSA key fingerprint is SHA256:72tL3r5bj1i3/m+Hh/kwuiy2fbksrLGsRj2XS60WQ6w.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

Type yes and hit return.

- ☐ At the prompt, enter the password: **n9sUs3r25**

This will copy the file *name.txt* across the network to the *Desktop* directory of the administrator machine. In the process, it will rename your file to your name with a *.txt* suffix. We will screen-share with the target computer to confirm the arrival of your file.

SCPing <u>from</u> a remote machine uses the syntax: 1) First, specify the required target file's location (machine-address:path-to-file/filename); 2) indicate where the file is to be saved on the local machine and what its name should be (path-to-file/filename).

```
scp username@remote.machine.address:source_directory/filename
target_directory/filename
```

In each case, transfer will begin upon entry of a valid password.

- ☐ Now let's grab a file from the Admin machine and (secure) copy it to your VM (using the same password as before). In this case, we are going to keep the existing filename, so we just need to tell scp where to put the file on the local machine (the "." in the command is a shortcut that means "here" in your current working directory):

```
scp ngs@35.171.154.104:Desktop/NGS-message.txt .
```

You just downloaded the *NGS-message.txt* file into your current (*unix*) directory.

- ☐ List the contents of the directory to ensure that it arrived:

```
•  ls .
```

- ☐ Note that we could have written the command without the period (i.e., **ls**), and we would get the same result. Try it.
- ☐ Use **cat** to read the contents of the *NGS-message.txt* file. (Remember, use tab completion.)

```
•  cat example/NGS-message.txt
```

1.8 Transferring files from a remote VM to your local machine

Some of the tools we will use in this workshop function best when run on our local machine. This will necessitate that we transfer copies of those files to our local computer. This can also be accomplished using **scp**. To learn how to do this, we will copy a sequence file (*MoRepeats.fasta*) from inside our *blast* directory to the PC (Mac) on which you are working:

- ☐ On your VM, list the *blast* directory to check that the file is where it is supposed to be. Here, we use an explicit path to the directory:

```
•  ls ~/blast
```

Or, we can use a relative path (assuming you are correctly located inside the *unix* directory):

```
•  ls ../blast
```

- ☐ Open a new terminal window. On a PC this can be done by typing "powershell" in the Start menu. Then, click on "Windows Powershell." On a Mac, open Spotlight search with command-spacebar, type Terminal, and then hit return. Or, if you're already using a terminal window, open another one by selecting "Shell" in the top menu bar and then choose "New Window."
- ☐ Copy the *MoRepeats.fasta* file from the *blast* directory on your VM to the current working directory on your local machine:

```
•  scp myName@XX.XXX.XXX.XX:blast/MoRepeats.fasta .
```

- ☐ Check that the file has been transferred successfully by listing it:

```
•  ls -l ~
```

- ☐ For further confirmation, you can also navigate to the file in your system's native file explorer. This can also be accomplished quickly from the command line.

On a PC:

- ☐ Open your home directory in File Explorer:

```
•  ii ~
```

ii	short for the Powershell command **Invoke-Item**

- ☐ Once the window opens, double-click on the file to open and read it.

On a Mac:

- ☐ Open your home directory in Finder:

```
•  open ~
```

- ☐ Once the window opens, double-click on the file to open and read it.

1.9 Downloading data from the internet

Normally, when you want to download a file from the internet, you open a browser, search for the file, and then download it by clinking on a link or an icon. How, then, do we download from the internet to a remote machine when we are not sitting in front of that machine, and we have no browser? Fortunately, there are command line tools that allow us to accomplish this task. Here, we will use a program called **wget** (web get) to download a file from the National Center for Biotechnology Information FTP site.

- ☐ In a web browser, open the page for *Saccharomyces cerevisiae* (bakers' yeast) at the NCBI website: (https://www.ncbi.nlm.nih.gov/genome/?term=Saccharomyces%20cerevisiae[Organism])
- ☐ Click on the top link that says **R64**. This is the main reference genome for *S. cerevisiae.*
- ☐ Click on the link that says FTP and then right-click on the selection that says *GCF_000146045.2_R64_genomic.fna.gz*
- ☐ Go back to your VM's terminal window
- ☐ **Make sure you are in the *unix* directory**
- ☐ Type "**wget** " (with a space afterwards), paste in the link that you just copied, specify an output filename with the `-O` flag (not a zero; it stands for "output"), and then hit return:

```
• wget https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/146/045/
  GCF_000146045.2_R64/GCF_000146045.2_R64_genomic.fna.gz -O
  example/yeast.nt.gz
```

Note: Our limited page width necessitates that we wrap long commands across multiple lines. **Do not hit return at the end of each line** because return tells the shell to execute whatever comes before it. **Therefore, all commands in boxes should be typed on a single line.**

Successful execution of the **wget** command should yield a runtime message similar to the following:

```
--2023-06-20 11:43:48--
https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/146/045/GCF_000146045.2_R
64/GCF_000146045.2_R64_genomic.fna.gz
Resolving ftp.ncbi.nlm.nih.gov (ftp.ncbi.nlm.nih.gov)... 165.112.9.229,
2607:f220:41e:250::7, 2607:f220:41e:250::11, ...
Connecting to ftp.ncbi.nlm.nih.gov
(ftp.ncbi.nlm.nih.gov)|165.112.9.229|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3843460 (3.7M) [application/x-gzip]
Saving to: 'yeast.nt.gz'

yeast.nt.gz
100%[=============================================================>]
3.67M  1.38MB/s    in 2.7s

2023-06-20     1:43:54 (1.38 MB/s) - 'yeast.nt.gz' saved [3843460/3843460]
```

- ☐ List the contents of your example directory to make sure you now have the yeast.nt.gz file.

```
•  ls example/
```

```
myName@ip-xxx-xx-xx-xxx:~/unix$ ls example/
Name.txt  testfile.txt  yeast.nt.gz
```

The new file shows up in red because it is a compressed archive.

If you do not see the file, check the download dialog. Were there any errors? Or did you forget to rename the file with the `-O` flag? Did you direct the output into the *example* directory?

A gzipped (compressed) archive was downloaded. It must be decompressed before we can look at it. But first, let's take a quick peek at the contents.

- ☐ We'll use **head** to look at the first 10 lines of the file (**use tab completion**).

```
•  head example/yeast.nt.gz
```

Note that the contents of gzipped files are not human-readable.

- ☐ So, let's decompress the file (**use tab completion**).

```
•  gunzip example/yeast.nt.gz
```

This will produce a file called *yeast.nt* in the *example* directory.

- ☐ List the *example* directory to check that the decompressed file exists.

```
•  ls example
```

- ☐ Let's have peek at the first 10 lines to check the decompression (**use tab completion**).

```
•  head example/yeast.nt
```

- ☐ You should see a "sequence header" line with information about the sequence, followed by lines with As, Cs, Gs and Ts—the actual DNA sequence data.

1.10 Copying and moving files

- ☐ Suppose we want to make a copy of our *yeast.nt* file in the same directory. We will need to give it a new name (**use tab completion for the old name**). Of course, the system doesn't yet know the new name, so tab completion won't work for that:

```
•  cp example/yeast.nt example/yeast_copy.nt
```

Note: the first argument of **cp** (copy) is the file you wish to copy, while the second argument is the destination and, if desired, a new name for the copy.

- ☐ List the *example* directory to check that a copy was made.
- ☐ Suppose we want to copy this file to a backup in a different directory and keep the same name. We just need to give the location for the copy. (Use tab completion.)

```
• cp example/yeast.nt extra/
```

- ☐ List the *extra* directory to check that the file was copied.

To move files from one location to another, we use the aptly named **mv** (move) program.

- ☐ We'll use **mv** to move the recently created *yeast_copy.nt* file from the *example* directory to the *extra* directory. (Use tab completion.)

```
• mv example/yeast_copy.nt extra/
```

- ☐ List the *extra* directory again to check that the file was copied.
- ☐ List the *example* directory, and you will see that the *yeast_copy.nt* is no longer there.
- ☐ Now we'll learn another useful feature of **mv**. We're going to move the *yeast_copy.nt* file back into the *example* directory but we're going to rename it at the same time. (Use tab completion for the existing file.)

```
• mv extra/yeast_copy.nt example/yeast_backup.nt
```

- ☐ Use **ls** again to check that the intended file was created.

1.11 Renaming files and directories

There isn't a dedicated rename command in UNIX. Instead, we can trick **mv** to do the rename for us. We just tell it to move the file to the same location under a different name.

- ☐ Use **mv** to rename the *yeast_backup.nt* file to *yeast_copy.fasta*. (Use tab completion for the original file.)

```
• mv example/yeast_backup.nt example/yeast_copy.fasta
```

- ☐ Use **ls** to check that the renaming was successful.

1.12 Removing files and directories

- ☐ Suppose we want to remove our *yeast_copy.fasta* file. We use **rm** (remove). (Use tab completion.)

```
• rm example/yeast_copy.fasta
```

List the contents of the *example* directory. The *yeast_copy.fasta* file is gone, permanently. There is no such thing as a "trash can" when dealing with the terminal/console.

- ☐ Removing entire directories has the potential to be disastrous because they could contain hundreds of precious files. Let's try it (Use tab completion):

```
• rm extra2
```

```
rm: cannot remove 'extra2': Is a directory
```

☐ Fortunately, **rm** won't remove a directory without the `-r` flag (recursive). (Use tab-completion.)

```
•  rm -r extra2
```

☐ List the contents of your current working directory. The *extra2* directory is also now gone.

Part II - The Bioinformatician's UNIX Toolbox

1.13 Redirection

- ☐ We will use some DNA sequence data to learn how to extract information from large files. Make sure you still have the *yeast.nt* file that we downloaded earlier. If not, use **wget** to re-download it from NCBI into your *unix/example* directory and be sure to unzip the file using **gunzip** before continuing.

- ☐ Let's make sure we are working in the *unix* directory.

```
•  cd ~/unix
```

The best way to understand redirection is to try it.

- ☐ Let's look at the first ten lines of the *yeast.nt* file.

```
•  head example/yeast.nt
```

It is sometimes useful to see more or fewer than the first ten lines of a file. We can specify how many lines we want to see using the -n option to **head**. For example, to show the first 20 lines of our *yeast.nt* file, we could use the command below.

```
head –n 20 example/yeast.nt
```

Or, if you're super lazy, you can also specify the number of lines using just "-":

```
head –20 example/yeast.nt
```

- ☐ Now we'll redirect the output to a file named *output.txt* in the *example* directory.

```
•  head example/yeast.nt > example/output.txt
```

- ☐ List the contents of the *example* directory. You should see an *output.txt* file.
- ☐ **cat** the *output.txt* file to display its contents. It contains the first 10 lines of the *yeast.nt* file
- ☐ Now try this:

```
•  cat < example/output.txt
```

We just used redirection to direct the *output.txt* file into **cat**. This is a bit silly because **cat** can handle filenames perfectly well without the "<" symbol, but the example serves to illustrate the point that the < and > symbols, respectively, can direct files into and out of UNIX programs. Some (not many) bioinformatics programs send the output to the screen and do not save results by default. In such cases, you can redirect the output to a file using the ">" symbol.

1.14 grep

Grep is a tool that can search for patterns within text files. It is based on regular expressions, a language for representing patterns. Patterns can get quite complex, but we'll stay relatively simple. Before we advance any further, we should make sure we understand the contents of the *output.txt* file which contains DNA sequence information.

- ☐ Use **cat** to display the contents of the *output.txt* file.

The first line,
`>NC_001133.9 Saccharomyces cerevisiae S288C chromosome I, complete sequence`
starts with the ">" symbol which tells us that this line contains the metadata for the DNA sequence that follows it. The `NC_001133.9` field is the official accession number for that sequence. The following text in the line provides additional information which tells us that this is the sequence for *Saccharomyces cerevisiae* (baker's yeast) strain S288C, chromosome I. The *output.txt* file does not contain the complete sequence, however, because we only extracted the first nine lines of sequence data.

First, we're going to search for every line of sequence metadata in the *yeast.nt* file. However, as we've just learned, the symbol used to define the metadata (>) happens to have a special meaning in UNIX—it means output the results of a command to a file. For this reason, we must be very careful when searching for ">" symbols.

The main pattern searching program in UNIX is **grep**; the command has the general format

```
grep <pattern> <filename>
```

which means search for this pattern in this file. If we were to search for the > symbol without doing anything to tell **grep** that we're looking for the actual symbol and not redirecting its output, e.g.

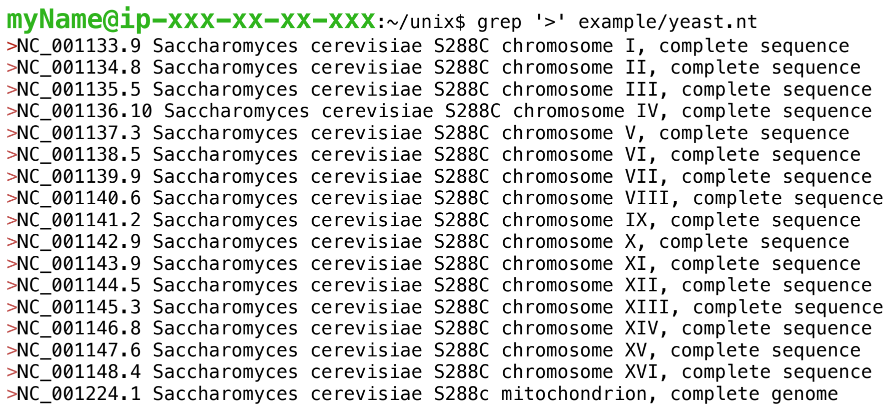
```
grep > example/yeast.nt
```

This will throw an error:

```
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.
```

This is because the shell interprets `grep > example/yeast.nt` to mean: search for nothing in no files and redirect the output to a file called *yeast.nt*. The end result is that it will overwrite our lovely *yeast.nt* sequence data with an empty output. So, instead, we need to inform **grep** to search for the ">" character by placing it in quotation marks:

- `grep '>' example/yeast.nt`



```
myName@ip-xxx-xx-xx-xxx:~/unix$ grep '>' example/yeast.nt
>NC_001133.9 Saccharomyces cerevisiae S288C chromosome I, complete sequence
>NC_001134.8 Saccharomyces cerevisiae S288C chromosome II, complete sequence
>NC_001135.5 Saccharomyces cerevisiae S288C chromosome III, complete sequence
>NC_001136.10 Saccharomyces cerevisiae S288C chromosome IV, complete sequence
>NC_001137.3 Saccharomyces cerevisiae S288C chromosome V, complete sequence
>NC_001138.5 Saccharomyces cerevisiae S288C chromosome VI, complete sequence
>NC_001139.9 Saccharomyces cerevisiae S288C chromosome VII, complete sequence
>NC_001140.6 Saccharomyces cerevisiae S288C chromosome VIII, complete sequence
>NC_001141.2 Saccharomyces cerevisiae S288C chromosome IX, complete sequence
>NC_001142.9 Saccharomyces cerevisiae S288C chromosome X, complete sequence
>NC_001143.9 Saccharomyces cerevisiae S288C chromosome XI, complete sequence
>NC_001144.5 Saccharomyces cerevisiae S288C chromosome XII, complete sequence
>NC_001145.3 Saccharomyces cerevisiae S288C chromosome XIII, complete sequence
>NC_001146.8 Saccharomyces cerevisiae S288C chromosome XIV, complete sequence
>NC_001147.6 Saccharomyces cerevisiae S288C chromosome XV, complete sequence
>NC_001148.4 Saccharomyces cerevisiae S288C chromosome XVI, complete sequence
>NC_001224.1 Saccharomyces cerevisiae S288c mitochondrion, complete genome
```

Re-run the previous command (use the up arrow to navigate back to it) and redirect the output to a file called *seqHeaderLines.txt* inside the *example* directory:

```
• grep '>' example/yeast.nt > example/seqHeaderLines.txt
```

- ☐ List the contents of your *example* directory. You should now see the *seqHeaderLines.txt* file.
- ☐ Use **cat** to display the contents of the *seqHeaderLines.txt* file.
- ☐ By adding the `-c` option to your **grep** command, you can use **grep** to count the number of lines that matched your search criteria. **Paired with the ">" pattern, this is a quick way to count how many sequence records are in a FASTA file.**

```
• grep -c '>' example/yeast.nt
```

- ☐ How many sequences does the *yeast.nt* file contain? ______________________
- ☐ Try this:

```
• grep '>' example/yeast.nt -A 5 -B 2
```

- ☐ Look at the output and guess what the -A and -B options do? ______________________. If you can't guess, read the manual page for grep to check the various options (man grep).

1.15 Pipes

It is possible to build small-scale workflows by sending the output of one program to the input of another.

- ☐ Let's grab the sequences header (metadata) lines again (**use the up-arrow key to go back to the earlier command**):

```
• grep '>' example/yeast.nt
```

- ☐ Now suppose we only want to see the headers of the first five sequences in the file.

```
• grep '>' example/yeast.nt | head -n 5
```

The | symbol is a "pipe" and acts as a middleman between **grep**'s output and **head**'s input. It says, use the output of **grep** as the input to **head**.

Note that grep also has a similar function: `grep -m 5 '>' example/yeast.nt` will accomplish the same thing (-m means max-count of matching lines)

- ☐ We could also count the number of matching lines by piping the **grep** output using another program called **wc** (word count):

```
• grep '>' example/yeast.nt | wc -l
```

`-l` **wc** returns the number of lines, words and characters in the input. -l means return only the number of lines.

Of course, the use of **wc** with **grep** is unnecessary because we already learned that **grep** has its own, built-in counter

- ☐ Now try this:

```
•  grep -v '>' example/yeast.nt
```

- ☐ With everything scrolling down the screen so fast, it's hard to see what's going on, isn't it? Hit control-c and try piping (|) the results through **head** to see the first ten lines of output. Compare the output of this command with the one from `grep '>' example/yeast.nt` and see if you can determine what the -v option does. Look at the manual page for **grep** to check your answer.

1.16 awk

awk is a programming language that is typically used for parsing and filtering/selecting text—it is especially useful for working on files with a tabular structure.

- ☐ Repeat the **grep** command to retrieve the header lines for each sequence in the *yeast.nt* file. The output will have lines that look like this:

  `>NC_001133.9 Saccharomyces cerevisiae S288C chromosome I, complete sequence`

- ☐ What if we wanted to filter these search results so that only the first part of the sequence header is displayed (i.e., everything before the first space)? We can create a sequence of piped commands to do this.

```
•  grep '>' example/yeast.nt | awk '{print $1}'
```

```
myName@ip-xxx-xx-xx-xxx:~/unix$ grep '>' example/yeast.nt | awk '{print
$1}'
>NC_001133.9
>NC_001134.8
>NC_001135.5
>NC_001136.10
>NC_001137.3
>NC_001138.5
>NC_001139.9
>NC_001140.6
>NC_001141.2
>NC_001142.9
>NC_001143.9
>NC_001144.5
>NC_001145.3
>NC_001146.8
>NC_001147.6
>NC_001148.4
>NC_001224.1
```

The part of the command in single quotes is **awk** code. This example has one line of code that prints a variable called `$1`. `$1` specifies the first column of each line, as determined by the table's field separators (which by default are spaces and tabs).

- ☐ Can you guess what substituting $1 with $2, $3, etc. would do? ______________________
  ____

- ☐ Try the respective commands to find out.

awk programming can get quite complicated. We could spend an entire session on it, but, for now, just be aware that it is useful whenever selection/filtering/reorganization of tabular data is needed.

- ☐ Instead of printing out the contents of specific fields, we could instead get the length of each line matched by **grep**.

```
•  grep '>' example/yeast.nt | awk '{print length($0)}'
```

- ☐ Or the length of a given field:

```
•  grep '>' example/yeast.nt | awk '{print length($1)}'
```

- ☐ One more particularly useful **awk** operation is shown below. (This is especially useful when working with long DNA sequences.) Run this command and look at the output to try and understand what it is doing.

```
•  grep -v '>' example/yeast.nt | awk '{print substr($1, 1, 10)}'
```

- ☐ I think the **substr** function does the following: ______________________
  Test your answer by playing around with the arguments provided to **substr** (try changing the values of $1, 1 and 10)

- ☐ Finally, note that we can search for strings using **awk** itself—we don't have to involve **grep** at all. In its most simple form, the structure of an **awk** command is as follows:

```
awk ‘check for some condition {do the following if condition is true}’ filename
```

- ☐ In the **awk** code below, `$1 ~ />/` indicates that **awk** should only run the code in braces if the lines match the pattern “>”. Note that we do not need to escape the “>” symbol here because the surrounding single quotes ensure that the symbol is interpreted literally.

```
•  awk '$1 ~ />/ {print substr($1, 1, 10)}' example/yeast.nt
```

- ☐ If we wanted to run code to search for lines that do not match the ">" pattern, we can use:
  `$1 !~ />/` instead. Try it…

- ☐ Lastly, let's use **awk** to perform some gymnastics on our header lines. Trying running this command:

```
•  awk '$1 ~ />/ {print $1, $3, $6 “Are we having fun?,”, $4, $2}'
   example/yeast.nt
```

Hopefully, you get a sense of how **awk** can be used for picking, choosing and reorganizing data in tabular formats

1.17 sed

Let's suppose we want to use the list of sequence headers we got from **awk** as input to a fictional script that only accepts numbers as input. How do we get rid of the >NC_ prefix? We could do this entirely with **awk (**using **substr)**, but we could also pipe the output from **awk** to **sed** instead.

sed is a "stream editor" that relies heavily on pattern matching (much like **grep** does). It is an incredibly powerful tool and takes much practice to master. We will give you some basic exposure to **sed** by showing how to use it for a common task—pattern substitution.

In its simplest form, a **sed** statement looks like this:

```
sed ‘s/term1/term2/g’ filename
```

In the command above, the "s" means substitution. The stuff between the first and second slashes (term1, in this example) is the pattern to match, and the stuff between the second and third slashes (term2) is what to substitute the matched pattern with. The **g** flag at the end means "global" so that every occurrence on a line is substituted. Without this flag, only the first instance on each line would be changed.

- ☐ To solve the problem stated at the start of this section, we can simply substitute the >NC_ prefix with nothing (an empty string).

```
• awk '$1 ~ />/ {print $1}' example/yeast.nt | sed 's/>NC_//'
```

The above examples give only a tiny taste of what **awk** and **sed** can do but will allow you to accomplish many bioinformatics tasks. Among many other things, you now know enough to extract (and rearrange) specific columns from large lists of data (we will practice this in a later module), and how to perform global find and replace in huge files that cannot be read into a typical text editing program (such as Word).