Tutorial: UNIX Basics

James B. Pease

Department of Biology, Wake Forest University

Updated: July 8, 2019

1. Install software

1.1. First, we need to ensure that we all have several important pieces of software installed.

1.2. **A plain text editor**: Microsoft Word, etc. are "rich-text editors" meaning that in addition to the letter characters themselves, Word documents store all kinds of formatting information that is hidden in their data structure. We will need to use a plain text editor so that we can see all of the raw data in our "plain-text" files. If you have one you already use and like, please go ahead. I recommend ***Geany***, which works on OSX, Windows, and Linux. Geany is available for download here https://www.geany.org/Download/Releases, or through most Linux software libraries.

1.3. **A UNIX Terminal**:

1.3.1. If you have Linux or MacOSX, these already have a UNIX Terminal program (it's usually called Terminal). On MacOSX, you can open Terminal via Spotlight (the search icon in the upper right) or by opening Utilities in Applications. Once the program is open and the icon appears in the Dock, I recommend clicking-and-holding on the icon and selecting "Keep in Dock" so that it's easy to re-open later.

1.3.2. If you have Windows, you should download and install MobaTek http://mobaxterm.mobatek.net/download-home-edition.html.
Click the green button on the right and follow the instructions to install. If you would rather use Putty or some other UNIX terminal, that's fine too.

1.4. **Seaview**: This is a biological sequence viewer, aligner, and phylogeny maker that is a little bit low-tech, but pretty easy to use.
http://doua.prabi.fr/software/seaview. Works on Linux, MacOSX, and Windows. Just download and follow installation instructions.

2. Why learn to use a computer like this?

2.1. It's a fair question. There are a lot of practicing biologists who never learn to use a computer at the command-line level. Why bother?

2.2. First, the latest and greatest software often does not come with a glossy interface. This is because it often takes much longer to program the interface itself than it does to create the software algorithms (the actual functional part). Scientists often want to make software that helps them answer their questions, and have little incentive and time to make interfaces. So learning to work this way will prepare you to use majority of cutting-edge bioinformatics software that is only available at the command line.

2.3. Second, this will open up a whole new set of possibilities in terms of how to actually use your computer. Most people are using about 10% of the capacity of their computers and often do repetitive tasks laboriously instead of learning how to make the computer do the work. Automate and "pipeline" processes also reduces manual errors in science. A simple example is the difference between trying to manually change the same word repeatedly throughout a document instead of using the Search/Replace All feature of most document editors.

2.4. Finally, it helps you connect with the data itself and understand how the bits and bytes actually are processed. Doing bioinformatics using all summarized and visualized data through graphical interfaces is a little like taking a trip on train. On a train, its efficient and everything is largely done for you, but the scenery can whiz by, you do not get an organic sense of the terrain, and you have to stick to the tracks and cannot pursue side paths. We are going to learn how to drive the off-road mountain bike. Riding your bike is more difficult at first, but you get to interact with the landscape, understand small but meaningful details, and you have ability to take your own path and asking new questions.

2.5. Ok. Hopefully you are convinced. Enough philosophizing. Let's dive in.

3. How to use this guide

3.1. This guide include a series of steps, numbered in a nested numeric outline style. This is so that if you have a question, you can say "I am stuck on step 1.1"

3.2. Frequently you will see the font change into more typewriter-like font (like below). This denotes a command that you will enter into the terminal.

3.2.1.
```
$commands are formatted like this
```

3.3. In this guide (and many like it) a "$" character will appear at the start of each command **you should not type this character**. Why put it there then? This is to show that there are no preceding spaces or denote if text were to appear before the prompt where you enter text.

3.4. Sometimes you will see multiple lines of commands as below.

3.4.1.
```
$this commands is very very very very very long \
and does not fit all on one line
```

3.5. Note the "backslash" \ character. This denotes where the same command continues onto the next line. You can literally type a \ and press Enter, then continue on the next line, or just ignore it and keep typing the command. Contrast this with the example below. However, a \ must be followed *immediately* by Enter. If you use a \ in the middle of a line, this has a different meaning.

3.5.1.
```
$this is the first command
$this is the second command
```

3.6. In the example above, there are two separate commands at two separate $ command prompts. This means you should enter the first command and press Enter (the Enter key will be capitalized). After the first command completes, enter the second command and press Enter.

3.6.1.

3.7. **UNIX is case-sensitive, always.** The commands below are treated as completely separate commands by UNIX.

3.7.1.
```
$somecommand
$Somecommand
$Some-Command
```

3.8. When we discuss key combinations like Ctrl+C, this means you should hold the Ctrl-key and press the C-key. Since this is conventional, I will always use the upper case letters to refer to the key. So Ctrl+C refers to holding the Ctrl-key and pressing the C-key, while Ctrl+Shift+C means holding the Ctrl-key and the Shift-Key then pressing the C-key. Ctrl+c (with a lower case "c") will not be used.

3.9. The commands in these tutorials are shown in a teletype monospace font, such as would appear on a older typewriter. These fonts are commonly used by programmers since the characters are all exactly the same width and so that characters are easily distinguishable. Note lower-case L (`l`) and a number 1 (`1`) can be similar, so observe the difference here. Note also that captial O (`O`) and zero (`0`) are quite different different.

3.10. The terms “directory” and “folder” are now effectively synonyms for the nested hierarchical organizational scheme by which files are organized in a computer system. Technically, a “folder” is a type of file directory not invented until the visual folder icons in Windows 95. I will use directory throughout these tutorials, but folder, in common parlance, means essentially the same thing.

3.11. UNIX users have a spoken language all their own and often a way commands are verbalized by special pronunciation in conversations. I will indicate this verbalization of UNIX in curly brackets with italics. For example, the command `ls` is said {*“list”*}, and the command `mkdir` is {*“make-dur”*}.

3.12. Questions will also appear in the text for you to check your conceptual and technical knowledge.

3.12.1. Example:

```
QUESTION #1: What are orthologs?
Sequences from individuals separated by speciations.
```

3.13. A final note: **Do not just copy and paste the commands into the Terminal.** You are learning a new language. Copying and pasting the commands would be like trying to learn Spanish by pasting English into an online translator and pasting the results into your homework. Typing in these commands manually character by character builds. . . well. . . character. You will make mistakes. It will be frustrating. Then you will improve and you will remember these commands better. An exception are long file paths and URLS. These are often copied and pasted by pros to prevent errors.

4. Opening a terminal

4.1. First open your **Terminal** window and create a new session.

4.2. Once you have an open terminal window, enter the following command:

4.2.1.
```
$echo "hello, world!"
```

4.2.2. Congratulations! You have officially (if not before now) been inducted into the UNIX Lifestyle. “hello, world!” is one of the oldest test command phrases, going back at least to 1972.

5. Home directory

5.1. You should now see a new prompt on the remote server. You are in your **home directory**, which is the default place you will log into every time. You can think of it a little bit like the “Documents” folder on MacOSX or Windows. This your basic user directory, under which all other subdirectories that contain your files will be located.

5.2. In order to get oriented a little more specifically, you should find out what your **present working directory** is (i.e., the folder that you are executing commands in).

5.2.1.
```
$pwd
```

5.2.2. This shows that you are in the directory `/home/USERNAME/` (or something like this). This is what is called a **path**, which gives the complete address of a location in the computer's file directory structure. Note that in UNIX, the file path is a text address of a folder location, with the folder levels separated by forward slashes "/". In this case, the path shown is an **absolute path**, since it tells the address back back to the **root** of the system file organization. The root is analogous to the `C:\` drive on a Windows computer. It is the basic directory for the *computer*, versus your *user* home directory. The absolute path of the root is simply `/`, all the user directories are in a directory located at path `/home`, and your user directory is located inside this directory at `/home/USERNAME`.

5.2.3. ✎ **QUESTION 1:** : If you created a folder inside you home folder called `apples` and created two folders inside `apples` called `oranges` and `pears`, then what are the absolute paths of the directories `oranges` and `pears`.

5.3. Run the following `ls` {*"list"*} command to the list files in the current directory.

5.3.1.
```
$ls
```

5.3.2. You shouldn't see anything (because there should not be files in your home directory).

5.4. Now a make a file, and check again.

5.4.1.
```
$touch emptyfile.txt
$ls
```

5.4.2. The `touch` command creates a file with no data inside it. Then you have something to list (`ls`) in your home directory.

5.5. Now lets make a directory with `mkdir` {*"make-dir"*} for this first lab.

5.5.1.
```
$mkdir lab01
```

5.5.2. Why `lab01` and not just `lab1`? In UNIX, when we get to ten directories, the sorted order of the directories would be `lab1, lab10, lab2, lab20, lab3` (since it sees the 1 character before 2 before 3). If we added a leading zero, then the order becomes `lab01, lab02, lab03, lab10, lab20, lab30`.

5.6. Now we will list the contents again, but with an extra part to the command.

5.6.1.
```
$ls -l
```

5.6.2. The `-l` is what is called an **argument**. Arguments refer to all required or optional additional paramters given at the command line after a program name. In this case, `-l` stands for "long" and so gives you a long-form description of the files. Since `-l` takes no additional paramters with it, it is known as a **flag**.

5.6.3. You can see the output of this command gives a lot more information. In particular notice that in the left column `lab01` has a `d` for "directory" in front of the `rwx` characters (those are file permissions, which we may discuss later). It also shows the user who owns the files (should be you), the last date of modification, and the size of the files (in bytes).

5.7. The alternative to a flag is an **option**, which does take a parameter. An example of an option for `ls` is:

5.7.1.
```
$ls -w 20
```

5.7.2. In the command above the `-w` option is used to specify the width of the output text on the screen to 20 characters. Notice that the output moves text onto separate lines once the list of file names goes over 20 characters.

5.8. Try this command:

5.8.1.
```
$ls -w 2000
```

5.9. Now try using `-w` without a parameter value.

5.9.1.
```
$ls -w
```

5.9.2. You should get an error that the "option requires an argument" meaning that you needed to provide a value for the width. Just telling the program "change the width to" does not make sense without a number.

5.10. There are also arguments that do not use a – prefix (like with `mkdir` above), others that use one – (like `-w` above), and some that use two dashes (`--width` can be used instead of `-w`).

5.11. To see the complete set of arguments for a program you can use one of two approaches:

5.11.1.
```
$ls --help
```

5.11.2. Now you can see all the (surprisingly numerous) options for the `ls` command and examples of how to use it. The `--help` flag is nearly universal in UNIX-based software. Some programs also will use `-h` interchangeably or to give more/less help text than `--help`.

5.12. The second method is to use the manual program to access the manual (sometimes just abbreviated as the "man-pages").

5.12.1.
```
$man ls
```

5.12.2. ✎ **QUESTION 2:** What does the `-h` flag in `ls` do?

5.12.3. Now once you run this program you are actually in a special environment that views text documents. The command prompt is gone, but don't panic! You can use the Up and Down arrow keys (or Page Up/Down) to scroll through the text. When you are ready to leave the manual, just press "q" (for quit).

5.13. Now you should be back in the home directory.

6. Moving around

6.1. Change directories into the new directory you just created using the `cd` (change directory) command.

6.1.1.
```
$cd lab01
```

6.2. Now use `pwd` to see where you are.

6.2.1.
```
$pwd
```

6.2.2. This should show a path of `/home/USERNAME/lab01` (or something like that).

6.3. If we want go back to the directory we were just in, which is the "**parent directory**." We use:

6.3.1.
```
$cd ..
```

6.4. This is just `cd` followed by a space (you have to use a space) followed by two periods `..` {*"dot-dot"*}. The `..` refers to the parent directory of wherever you are. So if you imagine the directory structure like a hierarchical tree, `..` is always the directory "above" or containing the current directory that you are in.

6.5. Now let's go back to the `lab01` directory.

6.5.1.
```
$cd lab01
```

6.6. Use `pwd` one more time to confirm you are back in the `lab01` directory.

6.6.1.
```
$pwd
```

7. File manipulation

7.1. Move the `emptyfile.txt` file you made earlier from the parent directory into the current directory using the `mv` {*"move/em-vee"*} command.

7.1.1.
```
$mv ../emptyfile.txt ./emptyfile.txt
```

7.1.2. Be careful here. The first argument should have two periods and the second only one. When you use a single period "`.`", this refers to the *current* directory, where `..` refers to the *parent* directory of the current directory. So this command tells it to move it from the parent directory to the current directory.

7.2. We can "rename" files in UNIX this way. I say "rename" because what we are actually doing is moving the file to a different address. It amounts to the same thing, but technically there is no "rename" function in UNIX, we just "move" the file to a different name.

7.2.1.
```
$mv emptyfile.txt newname.txt
```

7.2.2. Note that we excluded the `./` this time. UNIX will assume you are talking about files in the current directory if you do not specify otherwise.

7.3. Try the reverse operation using a different command structure:

7.3.1.
```
$mv /home/USERNAME/lab01/newname.txt \
/home/USERNAME/lab01/emptyfile.txt
```

7.3.2. ✎ **QUESTION 3:** Is the command the same or different with/without the "\" character? Why?

7.3.3. This performs the same function as the above command, but with a key difference. In the first one, we used the *local path* of the file and UNIX correctly assumed we meant files in the current directory. In the second command, we used the *absolute path* (or full path) of the files, which specifies their location back to the root (/). The second command could have been run from *any* directory and move the files to those exact locations. The first one would *only* work in the directory where the files were located.

7.4. We can also make copies of the files with the cp {*"copy/cee-pee"*}.

7.4.1. **IMPORTANT!** Unlike the operating system, if you mv or cp a file to a file path that already exists it will NOT check if there is a file there already and prompt you to ask if you want to overwrite the file. It will just overwrite the file. Use with caution.

7.4.2.
```
$cp emptyfile.txt tempfile.txt
$ls
```

7.4.3. This makes a new copy of `emptyfile.txt` called `tempfile.txt`.

7.5. Since we do not need `tempfile.txt`, we can just remove (delete) the file using the rm {*"rem/arr-em"*} command.

7.5.1. **WARNING!** Linux will not prompt you (usually) to double-check if you really want to delete the file. Use with caution.

7.5.2.
```
$rm tempfile.txt
$ls
```

7.5.3. You can see that now the file is gone. In this case, gone does not mean in the "recycling bin" or "trash" where it can be recovered. **In UNIX, deletion is permanent** (by default). So be careful.

7.6. Make the file again, but this time notice the different rm flag.

7.6.1.
```
$cp emptyfile.txt tempfile.txt
$rm -i tempfile.txt
```

7.6.2. See how it prompts you to make sure you want to remove it. That's because you used the -i flag for "interactive" mode. In the next section, we will learn how to use a command-line text editor and make this policy the default.

8. Making a text file

8.1. So we've just discovered that, by default, UNIX does not prompt you in the case of cp, mv, and rm overwrites and deletions. However, we also know now that there is a -i flag that exists for rm that does check. As it happens all three commands have this flag. We can set it up so that this is the default behavior without having to remember the -i flag.

8.2. We are going to edit a file that customizes how the command-line shell (the basic operating system environment) how to interpret commands (for your username only). First, we will want to make a backup in case anything goes wrong.

8.2.1.
```
$cp ~/.bashrc ~/.bashrc.backup
```

8.2.2. The ~ character is a shorthand for your home directory path. So you can use it in place of /home/USERNAME/ when writing file paths.

8.3. But how did that file get there? I thought my home directory was empty?

8.3.1. In UNIX, files that begin with `.` are hidden. When you run `ls` it does not show hidden files unless you use the "all" flag.

8.3.2.
```
$ls -a ~
```

8.3.3. Here, you ran `ls` with the "all" `-a` flag to show you all files including hidden files. You also `ls`-listed your home directory while still in the `lab01` directory as your current directory. So `ls` can list files in any folder while still currently in other folder.

8.3.4. You should now be able to see both `.bashrc` and `.bashrc.backup` in your home directory.

8.4. Now let's open the `.bashrc` using a plain-text editor called `nano`.

8.4.1.
```
$nano ~/.bashrc
```

8.4.2. You can see again we have left the command prompt and entered into the program's text-based interface. There is a toolbar at the bottom that shows you some commands. The ^ symbol refers to the Ctrl-key on your keyboard.

8.4.3. You should see some text on the screen (or it may be empty).

8.5. Use the Down key to move down to the bottom and press Enter to start a new line.

8.6. Add the following lines of text *exactly as written, including spaces*:

8.6.1.
```
alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -i'
```

8.7. Press Ctrl+O, then the Enter-key to "Output" the file to memory (i.e., "save" it).

8.8. Press Ctrl+X to "eXit" `nano`

8.9. Now log out (yes! log out) of the server by using the command.

8.9.1.
```
$exit
```

8.10. Log back in using the same process as before.

8.11. Now again navigate to the `lab01` directory, copy the file, and remove it.

8.11.1.
```
$cd lab01
$cp emptyfile.txt tempfile.txt
$rm tempfile.txt
```

8.11.2. If the above operation worked, you should now *by default* be prompted if you really want to delete this file. You should enter `yes` or just `y` and press the Enter-key to confirm. `n` or `no` or anything else will cancel the operation. This will undoubtedly be helpful. Anyone who uses UNIX has a story about a regrettable `rm`. There are even stores about an unfortunate web site entrepreneur or two who accidentally `rm`-ed their whole business website.

8.11.3. You also just learned a little bit about how to use `nano`, which is not a powerful text editor, but is simple to use.

9. Other ways to view text

9.1. Make another file in the `lab01` directory using `nano`.

9.1.1.
```
$nano data.txt
```

9.2. Once in the `nano` interface, enter the following text

9.2.1.
```
guanine
cytosine
adenine
thymine
```

9.2.2. Now output (Ctrl+O, then Enter-key) and exit (Ctrl+X).

9.3. List the files long-form and you should see that the file is there and has a non-zero size

9.3.1.
```
$ls -l
```

9.4. For short files, you can view the contents of any file in the standard terminal output (without modifying the file) using a number of commands. Try the following and observe the results.

9.4.1.
```
$cat data.txt
$head -n 1 data.txt
$tail -n 2 data.txt
```

9.4.2. The first one is `cat` (short for "catalog"), which displays the entire contents of the file. The second and third are `head` and `tail`, which display the first and last lines, respectively of a file. By default, `head` and `tail` display 10 lines. Here we have used the option `-n` (for "number") to specify only one or two lines.

9.5. `nano` is a simple text editor, but not efficient at previewing large files. For read-only (no modifying) access to files (particularly large ones) a program called `less` is recommended.

9.5.1.
```
$less data.txt
```

9.5.2. Once again, you have entered a text-based program interface. You can scroll up and down using the Up/Down or Page Up/Down keys. If you want to see all the commands for `less` press the h-key. To quit the help and to quit the program press the q-key.

9.6. Finally, let's look a powerful text filtering and searching tool call `grep` (**g**lobally search a **r**egular **e**xpression and **p**rint).

9.6.1.
```
$grep "ade" data.txt
```

9.6.2. You should see that `grep` output was simply the line `adenine`. This is because our search term was "ade", so `grep` will look through the file for any *lines* of the file (not words or characters) that match this pattern and output it.

9.7. Now try a different search string (a string is a programming term for a set of characters)

9.7.1.
```
$grep "nine" data.txt
```

9.7.2. Now you should have gotten two lines output `adenine` and `guanine` which both contain the string “nine” (the other two do not)

9.8. We can also output this text to a new file by using a **redirect**.

9.8.1.
```
$grep "nine" data.txt > nine.txt
$less nine.txt
```

9.8.2. The first command runs the `grep` command and then we use `>` (think of it like a little arrow) to “redirect” the output that would have gone to the screen into a different file. When you run `less` you should see that `nine.txt` only has the same content as the `grep` output that went to the screen in the previous example.

9.8.3. Do not confuse this with the $ markers at the begginning of lines. Remember those characters should not be typed, and only denote the command prompt itself.

9.9. Using grep and cat together with redirects can be useful.

9.9.1.
```
$grep "nine" data.txt > nine.txt
$grep "thy" data.txt > thy.txt
$cat nine.txt thy.txt > notcytosine.txt
$less notcytosine.txt
```

9.9.2. Here we used two grep calls to make a file containing the two lines containing “nine” and a separate for the one containing “thy”. Then we can cat these files together by running cat on both and redirecting the output to a new file.

9.10. We could have also counted how many matches are in each grep file using `grep`.

9.10.1.
```
$grep -c "nine" data.txt
$grep -c "thy" data.txt
$grep -c "n" data.txt
$wc -l nine.txt
$wc -l thy.txt
```

9.10.2. The first two commands run `grep` with the `-c` (“count”) flag turned on. This simply returns the number of lines that match the search string, *not* the number of times the string is matched.

9.10.3. The third command shows this difference. If you try to count the number of matches of the string “n,” `grep` returns 4. This is because although there are 6 “`n`”s in the text (2 in adenine and guanine), there are only 4 lines with `n`, so `grep` still returns 4.

9.11. We can also count the total number of lines (and words) in a file with the `wc` command (“word count”). Lines are counted using the `-l` flag.

9.11.1.
```
$wc -l nine.txt
$wc -l thy.txt
```

9.12. Let’s learn another useful command: `sort`

9.12.1.
```
$sort data.txt
$sort data.txt > data_sorted.txt
```

9.12.2. The first command will output to the screen the lines sorted in alphabetical order. The second will redirect the same to a new file.

10. The escape hatch

10.1. Let's learn a command that is not necessarily very useful at the moment, but will help illustrate a principle. So far we have executed commands on very small files where the result appears instantaneously. Sometimes it happens that you will execute the wrong command or a command gets "stuck" for a variety of reasons or exhibits some errors that do not terminate the program (usually called "warnings").

10.1.1. The `sleep` command tells the computer to do just that, to wait for a specified number of seconds.

10.1.2.
```
$sleep 2
```

10.1.3. You can see the command prompt waits for two seconds, then the program exits. This program does nothing on its own, but can be used to help time up programs used in a more complex series of programs.

10.1.4. Now run this:

10.1.5.
```
$sleep 600
```

10.1.6. Now you can see the program is going to wait for 600 seconds (10 minutes), during which time you cannot enter any additional commands at the command prompt. Not ideal.

10.1.7. Now press Ctrl+C. This will **C**ancel the program. This is fairly universal in UNIX at the command line when script are running. Note however that inside a program environment interace, like `less` or `nano`. Ctrl+C will not work to quit the program or stop it.

11. Downloading and Compression

11.1. We might want to download a file from the internet. To do so we can use a command called `wget` {*"double-u-get"*}, which is short for "web get."

11.1.1.
```
$wget \
ftp://hgdownload.soe.ucsc.edu/goldenPath/hg38/database/chromInfo.txt.gz
```

11.1.2. Verify the file downloaded and open the file with `nano`.

11.1.3.
```
$ls
$nano chromInfo.txt.gz
```

11.1.4. You will be prompted `"chromInfo.txt.gz" may be a binary file.  See it anyway?`. Press the Y-key and then the Enter-key.

11.1.5. ✎ **QUESTION 4:** Describe what you see.

11.1.6. That's because the file is a compressed file (similar to a "zip file") and not in raw-text form.

11.1.7. Use Ctrl+X to leave `nano`

11.2. The file is compressed by the GZIP compression algorithm. In order to decompress it we can use.

11.2.1.
```
$gunzip chromInfo.txt.gz
$ls
$nano chromInfo.txt
```

11.2.2. The first command decompressed the file (g-unzip), then list the files. You will see that `chromInfo.txt.gz` is now `chromInfo.txt`, which is about 21 kB in size. The file is now uncompressed and thus able to be viewed with `less`..

11.3. You can re-compress the file by using `gzip`.

11.3.1.
```
$gzip chromInfo.txt
$ls -l
```

11.3.2. The files name should now once again be `chromInfo.txt.gz`.

11.3.3. ✎ **QUESTION 5:** What are the sizes of the compressed and uncompressed files.

11.3.4. For a small file like this, compression does not matter much, but consider a huge file of hundreds of GB (gigabytes). At the same reduction factor, we now only have to download a 10-GB file instead of a 70-GB file, which saves a lot of time and internet bandwidth.

12. Working with Text Files

12.1. We can search files using `less` to find information that we need.

12.2. Uncompress the again file by using `gunzip` and then view with `less`. However, this time when you are typing the name, type `gunz` and then press the Tab-key. You should see that the word "autocompletes" because there is only one command that starts with `gunz`. Now begin typing `chrom` and press the Tab-key again. Since there is only one file with a name starting with `chrom`, the name is autocompleted.

12.2.1.
```
$gunzip chromInfo.txt.gz
$less chromInfo.txt
```

12.2.2. Autocomplete saves a lot of time and frustration, practice using it as often as possible.

12.2.3. You can see that the file contains lines of text with three values each separated by "tab" characters. A tab-character is an invisible character that is *different from a space*. Tabs cue the text viewer to shift the text over to line it up. This type of file is called a tab-separated values (TSV) file. Where tab-characters are the "field delimeter" (i.e., the character between the data fields.

12.2.4. ✎ **QUESTION 6:** : What would you guess line 1 would look like in another comma data file type called a comma-separated values (CSV) file?

12.2.5. The values in file are the chromosome name, chromosome length, and some other information about the file formatting (2bit).

12.2.6. We can use the search function in `less` to find text quickly.

12.2.7. Press the "/"-key (forward-slash). You will see a prompt activate at the bottom of the screen.

12.2.8. Type `chrX` and press the Enter-key.

12.2.9. The less view window should advance so that the line starting with `chrX` is the top line and the characters `chrX` are highlighted.

12.2.10. The `/`-key activates the forward-search, which looks for the next match to you search string from your current position.

12.3. To backward-search, press Shift+/ (for ?), again a prompt will appear. Type `chr` and press the Enter-key.

12.3.1. This time less searched backward from your current position to find the previous match of the `chr` string, which should be the previous line.

12.3.2. ✎ **QUESTION 7:** Describe how to search for the mitochondrial chromosome `chrM`. What is the length of the mitochondrial reference chromosome?

13. ✎ QUESTION 8: Synthesis (record all steps)

13.1. Use the commands you've learned to do the following:

13.2. Download the file `ftp://hgdownload.soe.ucsc.edu/goldenPath/hg38/database/tRNAs.txt.gz`

13.3. Using `grep` and redirect to make a file that only shows tRNA sequences on the X chromosome.

13.4. Using `wc`, how many entries are there on the X chromosome?

13.5. Using the search in `less`, where is the tRNA gene labeled `tRNA-Ile-GAT-1-1` located on the X? (You should see two numerical columns that show the start and end coordinates)

13.6. Now rename the original tRNA file a hidden file using the `mv` command. What command did you use?

13.7. What command would you use to compress the output from `grep`?

14. ADVANCED EXERCISES (on your own, if you want extra practice. You will need to search the internet for help)

14.1. ✎ **QUESTION 9:** What commands using `tr`, `awk`, and/or `sed` to swap the first and second columns of `chromInfo.txt`. Upload the output as `chromInfoTranspose.txt`.

14.2. ✎ **QUESTION 10:** How can you use piping ("|") to take the output from a `grep` search and `sort` it all in one command? Include your command and the output file as `PipedOutput.txt`.

14.3. ✎ **QUESTION 11:** Create a shell script (.sh) file to execute any block of above commands in one call. Upload this file as `ShellScript.sh`.

14.4. ✎ **QUESTION 12:** Start a `sleep` process, "suspend" it, now make it run in the "background." Use the `ps` command to find the process, then use `kill` to end it early.

14.5. ✎ **QUESTION 13:** Execute a `grep` search on `chromInfo.txt` using any "regular expressions" record your command and output.