Introduction to UNIX

written by Jason Banfelder and Luce Skrabanek

September 1st, 2020

```
You see, in the UNIX world, 'system' means 'a bunch of unrelated programs'.
                                -- Steve Strassmann
```

1 Introduction (logging in, passwords)

1.1 Lecture

1. Introduction to UNIX
   (a) UNIX is the dominant operating system for high-performance, scientific computing.
   (b) Brief history
      i. UNIX was originally developed at AT&T Bell Labs in 1969.
      ii. There are now many flavors of UNIX. Some common aliases are: IRIX, Solaris, AIX, HP-UX, BSD, Linux (Red Hat, SUSE, Debian, Mandrake, etc), Mac OS X.
      iii. There are a couple of different UNIX shells. They all fall into two families, the Bourne shell family and the C shell family. Whenever possible, use the **bash** or **tcsh** shells. These are the latest and greatest members of the two families; others are lesser forms of these.
         A. The **tcsh** shell has historically been preferred by scientists.
         B. The **bash** shell is preferred by computer geeks. Traditionally, scripts are written in **sh** or **bash**. We will be using bash in this class.
2. Particulars of this class
   (a) If you are using an Apple laptop, it is running UNIX.
   (b) We want you to have the experience of working on a large UNIX server.
   (c) We will connect remotely to a server at Rockefeller University.
      i. Start the Terminal application from the Dock (or putty, if using a Windows machine).
      ii. Directions to connect to your account are given with your username and password.
      iii. Type your password.
   (d) A short discussion about security.
      i. Change your password using **passwd**.
   (e) To log out, use the **exit** command.

1.2 Exercise

1. Log in to your account using **ssh**.
   (a) **ssh eureka.rockefeller.edu -l** *<account name>*
2. Change your password using **passwd**.
3. Log out using **exit**.

2 Looking at Files (`ls, cat, more, head, tail`)

2.1 Lecture

1. **The UNIX way: (almost) everything is a file.**
   (a) Use the **`ls`** command to see what files you have.
      i. Example: **`ls`**
   (b) Common UNIX commands are usually a just few letters long. This savz kystrks. Although this makes UNIX appear cryptic and a little harder to learn, you will find you'll get to know these commands quickly. In the long run, you will be much more efficient.
   (c) What is in the file?
      i. Type **`cat quotation`** to see a pithy quotation. Note that this is a different file from **`Quotation`**, because case matters in UNIX.
   (d) File naming conventions in UNIX differ from Windows in several ways.
2. When you're typing commands, you can use the following:
   (a) The left and right arrow keys move the cursor left and right (surprising, eh?)
   (b) The up and down arrow keys scroll forward and backward in your history of commands. This is useful if you need to type a command that is similar to one you previously ran.
   (c) ⟨TAB⟩ autocompletes a (partial) unambiguous filename.
   (d) CTRL-A moves to the beginning of the line.
   (e) CTRL-E moves to the end of the line.
   (f) CTRL-D deletes the character that the cursor is on.
   (g) ⟨Delete⟩ works as expected.
   (h) CTRL-C abandons the whole affair and lets you try again.

2.2 Exercise

1. Log in again.
   (a) Did you remember your password? If not, you have failed this class. Try to leave unobtrusively at the next coffee break.
2. Looking at your files:
   (a) **`ls`**
   (b) **`cat quotation`**
   (c) **`cat Quotation`**
3. Looking at the instructor's files:
   (a) **`ls /ru-auth/local/home/unixinst/`**
   (b) **`cat /ru-auth/local/home/unixinst/quotation`**
   (c) **`cat /ru-auth/local/home/unixinst/Quotation`**
4. Looking at a classmate's files:
   (a) **`cat /ru-auth/local/home/unixst01/quotation`**
5. Looking at your files (another way):
   (a) **`cat /ru-auth/local/home/`***`<your username>`***`/quotation`** (note it is same as 2b)
6. Looking at big files:
   (a) **`cat /ru-auth/local/home/unixinst/dante/inferno.txt`**
   (b) **`more /ru-auth/local/home/unixinst/dante/inferno.txt`**
      i. To advance a line at a time, press ⟨Return⟩.
      ii. To advance a screenful, press the space bar.
      iii. When you just can't take it any more, press **`q`** to quit the pager.
      iv. Some UNIX systems have a better pager called **`less`** ("less is more" . . . *groan* . . . which lets you scroll up too) but it is not on all UNIX systems.

(c) **`head /ru-auth/local/home/unixinst/dante/inferno.txt`**
   i. This shows the first 10 lines of Dante's Inferno.

(d) **`head -n 20 /ru-auth/local/home/unixinst/dante/inferno.txt`**
   i. This shows the first 20 lines of Dante's Inferno.
   ii. **The UNIX way: command options are preceded by a `-`**

(e) Can you guess how to show the last ten lines?
   i. HINT: the opposite of **`head`** is . . . **`tail`** (very good!).

(f) **`tail /ru-auth/local/home/unixinst/dante/inferno.txt`**
   i. This shows the last 10 lines.
   ii. Can you guess what **`tail -n 20`** does?

3 Directories (`pwd`, `cd`, relative and absolute pathnames)

3.1 Lecture + Exercise

1. Aren't you sick and tired of typing **`/ru-auth/local/home/unixinst/`**? Didn't you say UNIX was all about svng kystrks?
2. Directories are hierarchical, akin to Mac or PC files and folders. In UNIX, the **`/`** separates the folder names from the file name.
3. There are no drive letters of volume names. The root directory is just **`/`**.
4. The current directory is **`.`**
5. The parent directory is **`..`**
6. To find out what your current directory is, use the **`pwd`** command.
7. To change your present working directory, use the **`cd`** command.
   (a) **`cd`** by itself is shorthand for going directly to your home directory.
   (b) **`cd /ru-auth/local/home/unixinst`** goes to the instructor's home directory.
   (c) **`cd /ru-auth`** followed by **`cd local`** followed by **`cd home`** and then **`cd unixinst`** is another way to connect to the instructor's home directory. Observe the difference between absolute and relative pathnames.
   (d) **`cd ../unixst01`** goes to the first student's home directory, if you are currently in your home directory.
   (e) **`cd`** ~**`unixst02`** is shorthand to connect to the second student's home directory.
   (f) What do you think **`tail -n 25`** ~**`unixinst/dante/inferno.txt`** would do?
8. Can you list all of the home directories on our server?
   (a) **`cd /ru-auth/local/home/`** and then **`ls`**.
   (b) Note that it is typical for UNIX systems to have hundreds of users, unlike PCs.

4 Manipulating Files and Directories (`cp, mkdir, mv`)

4.1 Lecture

1. We can now see what files we have and look at their contents. Next we will learn how to move files around in the directory structure.
2. Moving files around is equivalent to dragging and dropping files on your desktop computer. However, since we can't use a mouse with UNIX, we need some new commands.
   (a) To copy a file, use **`cp`**.
   (b) To copy the human `.pdb` file from the instructor's account, use
      i. **`cp ~unixinst/sequences/1A3B.pdb .`**
      ii. Make sure you're in your home directory before you do this. The period indicates that the file will be copied with the same name to the current directory.
3. Let's create a directory to put our newly acquired file into. We do this with the **`mkdir`** command.
   (a) **`mkdir human_data`**
4. Let's move our `1A3B.pdb` file into the new `human_data` directory.
   (a) **`mv 1A3B.pdb human_data`**
   (b) We can also rename files using the **`mv`** command
      i. **`mv 1A3B.pdb 1A3B_human.pdb.`** What do you need to do before you can execute this command?
      ii. HINT: what's your `pwd`; where's your file?
   (c) Can you think of a way in which you could have renamed the file without **`cd`**ing into the directory?
      i. **`mv human_data/1A3B.pdb human_data/1A3B_human.pdb`**
   (d) **`mv human_data/1A3B.pdb 1A3B_human.pdb`** would have renamed the file and moved it up into the current directory.
5. Note that you can also rename a file while you move it. How can you rename a file while copying it to the `human_data` directory?
   (a) **`cp ~unixinst/sequences/1A3B.pdb human_data/1A3B_duplicate.pdb`**
6. You can also move multiple files at the same time.
   (a) **`mv thrombin_human_aa.fasta thrombin_human.sw human_data`**
   (b) Note that when you are moving files this way, the destination must be a directory, and you can't rename files while you're moving them.
7. You can also move and rename directories with the same commands.
8. **The UNIX way:**
   (a) There are many ways of doing the same thing.
   (b) It is often possible to achieve a lot on one command line. This can rather cryptic but can be powerful, and less tedious.

4.2 Exercise

1. Copy all the data files from the instructor's sequences directory, make separate directories for each organism, move the relevant files to their proper directories and separate sequence files from pdb files. Check what organisms the PDB files come from. See the slide for a sample directory structure.
   (a) How few commands can you manage to do this in?

5 Introduction to Editing File Content - Part I (`vi`)

5.1 Lecture

1. Now let's learn how to manipulate the contents of our files. There are many file (or text) editors available in UNIX, but not all systems have all of them. The one editor that is always available is **`vi`**. This is not a word processor, and is not completely intuitive from the start. This is going to be the hardest part of the course, and if you can get through this, you'll find the rest plain sailing.
2. **`vi myfile`**
   (a) This creates the file and opens it for editing if it does not already exist; if it does exist, it is opened for editing.
3. vi has three modes: a command mode, a surfing mode and an input mode.
   (a) When you start **`vi`**, you're in the surfing mode. This means that you can move around the file, and delete text. When you're in surfing mode, the keys you press will not show up on the screen, but will rather be interpreted as instructions to move around the screen, delete text or switch to either of the other two modes.
   (b) Since we want to enter text, let's learn one way to switch to input mode.
      i. Press **`a`**. This stands for 'append'. Anything you type now will appear in the file. To get out of this mode, hit ⟨Escape⟩.
   (c) We use command mode to issue more complex commands, such as saving the file, and quitting out of **`vi`** to the shell again. You enter the command mode by pressing the colon key, typing the command, and pressing ⟨Return⟩. To quit **`vi`** without saving your work, type **`:q!`** and press ⟨Return⟩. To save your work, type **`:w`**. To save and quit at the same time, use **`:wq`**. Don't forget to press ⟨Return⟩ after entering these commands!
4. Note: sometimes it is NOT obvious from looking at the screen which mode you are currently in. If you are not sure, hit ⟨Escape⟩ a couple of times to make sure you are in surfing mode. The computer may beep at you, but it's okay ... even the best UNIX gurus do this when they type too fast and their brain has trouble keeping up with their fingers.

5.2 Exercise

1. If at any point you get into trouble and panic, hit ⟨Escape⟩ a couple of times, then type **`:q!`**, press ⟨Return⟩ and start again.
2. Create a README file in one of your newly created directories. Describe what's in that directory. Since we haven't yet taught you how to edit the file, don't worry about mistakes and typos just yet. (For this exercise, we will tell you every keystroke and exactly when to press ⟨Return⟩!)
   (a) **`vi README`** ⟨Return⟩
      i. Since the README file does not already exist, it is created.
   (b) **`a`**
      i. Do not press ⟨Return⟩ you are in input mode as soon as you press the **`a`** key.
   (c) **`This is a directory that contains human sequence data.`** ⟨Return⟩
      i. Here the ⟨Return⟩ keystroke creates a new line in the file. You are still in input mode.
   (d) **`The sequences are in fasta and swissprot formats.`**
   (e) ⟨Escape⟩
      i. You have just "escaped" from input mode.
   (f) **`:wq`** ⟨Return⟩
      i. You have saved your file and exited **`vi`**!
      ii. Phew!

6 Introduction to Editing File Content - Part II (`vi`)

6.1 Lecture

1. Let's learn how to edit the file we just created. We will change the case of the words fasta and swissprot, and make explicit that the sequences are protein.
2. To move around, use
   (a) **`k`** up
   (b) **`j`** down
   (c) **`h`** left
   (d) **`l`** right or ⟨SPACE⟩
   (e) Note that these keys are all beside each other on the keyboard. They don't stand for anything; they're just easy to get to. Game players may be well used to using these keys!
   (f) In our flavor of UNIX, you can also use the arrow keys to move around. This is not always the case, and real UNIX geeks use the h, j, k, and l keys because they always work on all systems.
3. Move the cursor to the space between the word 'human' and the word 'sequence'. Pressing **`a`** now will switch you to input mode. Type the word `protein` followed by a space. Now press ⟨Escape⟩ to leave input mode and return to surfing mode.
4. Now move the cursor onto the 'f' of 'fasta'. Pressing **`x`** will remove the character under the cursor. Press **`x`** four more times to delete the whole word.
5. Now insert the word `FASTA` in uppercase. This time, to enter input mode, press **`i`**. This stands for 'insert' (as opposed to 'append'). The difference between insert and append is that insert starts adding at the position under the cursor, append starts adding at the position following the cursor.
   (a) Often if doesn't matter which one you use so long as the cursor is in the right place. But:
      i. If you want to add a character before the first character of a line, you must use **`i`**.
      ii. Similarly, to add a character after the last character of a line, you must use **`a`**.
6. Correct the spelling of swissprot to SwissProt by moving the cursor over the first 's', pressing **`x`** to delete the lowercase 's', pressing **`i`** to enter input mode, and then typing `S` to insert an uppercase S. This is followed by ⟨Escape⟩ to return to surfing mode. Follow the same steps to change the 'p' into 'P'.
7. Save the file.

6.2 Exercise

1. Make these changes and correct any typos you may have made.
2. Copy the human `README` file to the rodent and bovine directories and replace the organism name as appropriate. DO NOT RETYPE THE WHOLE FILE.

7 Deleting Files and Permissions (`rm, rmdir, chmod`)

7.1 Lecture

1. You can remove files using the **`rm`** command. There is NO UNDO. If you remove a file, you're not getting it back.
   (a) To illustrate this, let's copy a file that we can remove.
      i. **`cp README DELETEME`**
      ii. **`rm DELETEME`**
2. To delete a directory, use the **`rmdir`** command. Note that the directory must first be empty.
3. You saw before that UNIX is a multi-user operating system. There needs to be a mechanism in place to ensure that you don't corrupt (or delete) other people's files (even if you can see and read them), and that other people don't corrupt (or delete) yours. To do this, each file and directory has a series of permissions associated with it. This tells the system which people can and can't read, write and execute those files.
4. We've already learnt about the **`ls`** command. The **`ls`** command has a **`-l`** option which shows you a lot of information about the files and directories, including the permissions associated with them.
5. Let's try to understand the output from **`ls -l`**?

```
drwxr-xr-x 2 unixinst unixcls 4096 Aug 31 15:01 artists
drwxr-xr-x 4 unixinst unixcls 4096 Aug 31 15:01 bin
drwxr-xr-x 2 unixinst unixcls 4096 Aug 31 15:01 blast
drwxr-xr-x 3 unixinst unixcls 4096 Aug 31 15:01 build
drwxr-xr-x 2 unixinst unixcls 4096 Aug 31 15:01 csplit
drwxr-xr-x 2 unixinst unixcls 4096 Aug 31 15:01 dante
drwxr-xr-x 2 unixinst unixcls 4096 Aug 31 15:01 downloads
drwxr-xr-x 2 unixinst unixcls 4096 Aug 31 15:01 examples
drwxr-xr-x 2 unixinst unixcls 4096 Aug 31 15:01 nimblegen
drwxr-xr-x 2 unixinst unixcls 4096 Jan 22  2017 perl5
-rw-r--r-- 1 unixinst unixcls  132 Oct 29  2007 quotation
-rw-r--r-- 1 unixinst unixcls   81 Apr  6  2004 Quotation
drwxr-xr-x 2 unixinst unixcls 4096 Aug 31 15:01 sequences
drwxr-xr-x 6 unixinst unixcls 4096 Aug 31 15:01 sqlite3
-r--r--r-- 1 unixinst unixcls 1966 Oct 29  2008 stairway.txt
```

6. The last column is the filename or directory name, which is what appears when you do an ordinary **`ls`** command. Preceding that, is the date and time at which that file or directory was last modified. Continuing to work backwards, we see the size of the file or directory in bytes. The next column is the group owner (we'll say more about this in a moment). Then we have the username of the owner of the file. If you own the file, this will be your username.
7. At the beginning of the line are a series of 10 letters or hyphens which show the filetype and permissions. The first character gives the file type: directories are shown with a `d` and regular files with a hyphen. The remaining letters are `r`, `w`, and `x` and represent 'read', 'write' and 'execute' permissions, respectively. There are three sets of these letters:
   (a) The first set shows the permissions that the owner himself has for this file.
   (b) The second set indicates the permissions for the group of users that the user belongs to.
   (c) The third set shows the permissions for all other users who have accounts on the system.
8. You can specify who can read, write or execute your files by using the **`chmod`** command. The user is referred to by `u`, the group by `g` and all other users by `o`. You can add, or take away, permissions using `+` and `-` so long as you are the owner of the file.
   (a) Make your quotation file readable by you only.

   i. **`chmod go-r quotation`** (you should be in your home directory)
   ii. Note that the system administrator can read anybody's file so this is not a solution to true privacy.

(b) Make the same file readable and writable by everybody.

   i. **`chmod go+rw quotation`**

9. Users can belong to more than one group, which makes permissions slightly more complicated than described above, but we're not going to worry about that.

8 Some Cool Stuff

8.1 Lecture

1. Cool command line tricks:
   (a) You can use a semi-colon to type multiple commands on the same line. This can be especially useful when you are typing the same group of commands over and over again, so you can just use the up arrow to rerun those commands.
      i. **mkdir** *<directory-name>***; cd** *<directory-name>*
   (b) When you type a command, the combination **!$** is replaced by the last word of the previous command line. This is usually used when the last word of the preceding command is a filename.
      i. **wc -l** ~**unixinst/quotation**
      ii. **cat !$**
      iii. But **mkdir** *<directory-name>***; cd !$** does not work.
2. **The UNIX way: each command is a tool in your toolbox. You combine them to build what you need. Once you know a critical mass of commands, UNIX becomes more than the sum of its parts because you can combine commands in an infinite variety of ways.**
   (a) UNIX rewards cleverness, so don't be afraid to be clever (but make sure your stuff is backed up! UNIX can be a powerful weapon, but you can still shoot yourself in the foot with it).
3. The **wc** command (word count) counts lines, words, and characters in a file.
   (a) **cat quotation**

```
THERE IS NOTHING NOBLE IN BEING SUPERIOR TO SOME OTHER
MAN. TRUE NOBILITY IS BEING SUPERIOR TO YOUR FORMER SELF.
  -- HINDU PROVERB
```

   (b) **wc quotation**

```
3 23 132 quotation
```

   (c) This reports 3 lines, 23 "words", and 132 characters in this proverb.
      i. Count the words yourself. Did you get 23? Why do you think the **wc** command reports 23 words? What is the definition of a word that the **wc** command uses?
4. To learn more about a command, use the **man** command.
   (a) **man wc**
   (b) This tells you everything you wanted to know about a command, and a lot more besides ...
   (c) The "man pages" can be a bit dry, but are usually very precise.
   (d) When the output of the **man** command is longer than a screenful, it is "piped" into the less command, so you can view it one page at a time.
      i. Some UNIX flavors pipe the output into the **more** command.
5. **The UNIX way: The idea of "piping" the output of one command to the input of another is a key concept in getting the most out of UNIX.**
   (a) The **man** command pipes its output to the **less** command silently ...
   (b) Usually we need to specify this behavior explicitly. This is done with the **|** operator. UNIX geeks call it the "pipe" character.
   (c) We can use pipes to figure out how many words there are in the first 25 lines of Dante's Inferno.
      i. Step 1: What are the first 25 lines of Dante's Inferno?
      head -n 25 ~**unixinst/dante/inferno.txt**
      ii. Step 2: How many words are in the result of the previous command?
      wc -w
      iii. Steps 1 & 2 together!!!
      head -n 25 ~**unixinst/dante/inferno.txt | wc -w**
   (d) The more commands you know, the better off you are. Your capabilities grow exponentially as does your ability to combine the commands you know.

9 Introduction to Pattern Matching (`egrep`)

9.1 Lecture

1. The **`egrep`** command is used to search files for patterns.
   (a) **`more 1TAW.pdb`**
      i. Note that each line of a PDB file begins with a keyword that describes the information on that line.
   (b) To show only the TITLE lines:
      i. **`egrep TITLE 1TAW.pdb`**
   (c) In the above example, we are searching for a specific string (TITLE). You can also look for more general patterns. UNIX geeks call these patterns "regular expressions". These regular expressions (or REs) can appear cryptic, but they are **very** powerful. There are entire books on regular expressions. We will not go into quite that much detail, but will spend some time on them because they are so useful.
   (d) The simplest RE is an exact textual match of a word (as we saw above when looking for the TITLE lines). But this can get you into trouble ...
      i. How many atoms are there in the 1TAW structure?
         A. **`egrep ATOM 1TAW.pdb | wc -l`**
         B. Congratulations: you just shot yourself in the foot. Why? Try: **`egrep ATOM 1TAW.pdb | less`**
      ii. What you need to do is search for all of the lines that begin with ATOM (not just that contain ATOM anywhere on the line).
         A. **`egrep '^ATOM' 1TAW.pdb | wc -l`**
         B. **`egrep -c '^ATOM' 1TAW.pdb`**
      iii. You can also try:
         A. **`egrep -v '^ATOM' 1TAW.pdb | less`** to ensure that there are no false negatives.

9.2 Exercise

1. Count the number of residues in the human integrin/collagen complex structure.
   (a) **`egrep '^ATOM' 1DZI.pdb | egrep CA | wc -l`** happens to work, but
   (b) **`egrep '^ATOM' 1DZI.pdb | egrep ' CA ' | wc -l`** is a little more precise. We include spaces around the `CA` to ensure that we find only whole words.
2. Count the number of residues in each chain.
   (a) **`egrep '^ATOM' 1DZI.pdb | egrep ' CA ' | egrep ' A ' | wc -l`** works fine, but ...
   (b) What happens when you try to count the residues in chain C?
      i. Don't worry about solving this problem right now.
3. Count the number of glycines in chain B.
   (a) **`egrep '^ATOM' 1DZI.pdb | egrep ' CA ' | egrep ' GLY B ' | wc -l`**

10 Redirection and Advanced Pattern Matching (**egrep**)

10.1 Lecture

1. Frequently you want to save the results of your commands to a file. This is like piping, except that the output goes to a file instead of being used as the input to another command. Because the output goes to the file, you don't see it on the screen.
2. Use the **>** operator to indicate that you want your results to be redirected to a file.
   (a) Save the lines containing PRO residues in chain B to a file.
      i. **egrep '^ATOM' 1DZI.pdb | egrep ' CA ' | egrep ' PRO B ' > x**
      ii. **cat x**
   (b) What happens when there is an error in your command?
      i. **egrep '^ATOM' 1dzi.pdb | egrep ' CA ' | egrep ' PRO B ' > x**
      ii. **cat x**
      iii. Note that the errors are not saved to the file (**x** is empty) but still appear on the screen. This is because *stderr* is different from *stdout*.
   (c) To save all of the results (i.e., both *stderr* or *stdout*) to the file, use ...
      i. **egrep '^ATOM' 1dzi.pdb > x 2>&1**
   (d) To append results to a file, use ...
      i. **egrep '^ATOM' 1DZI.pdb >> x**
   (e) See the accompanying slides for an explanation of *stdin*, *stdout*, and *stderr*.
3. More **egrep** patterns:
   (a) **.** matches any single character.
   (b) **()** group the enclosed items into a single item.
   (c) ***** matches zero or more occurrences of the preceding item.
   (d) **+** matches the preceding item one or more times.
   (e) **?** matches the preceding item zero or one times.
   (f) ^ matches at the beginning of a line only.
   (g) $ matches at the end of a line only.
   (h) **[]** matches any one occurrence of the characters enclosed within the brackets. If a hyphen is included, this indicates a range. All of the above special characters lose their special meaning within the brackets. ^ takes on another special meaning within these brackets. If ^ is the first character within the brackets, this indicates that the string should match any character except those in brackets.
   (i) **{m}** matches m occurrences of the preceding item.
   (j) **{m,n}** matches anywhere between m and n occurrences of the preceding item.
   (k) \ makes all of the above special characters lose their special meaning (including itself). This is called "escaping out" the special character. Its special effect is also lost within brackets, as above.
4. **egrep** command options
   (a) Different implementations of UNIX have varying **egrep** options. Some of the most ubiquitous and important are:
      i. **egrep -v**
      Reverses the sense of the match. Prints lines that don't match the specified pattern.
      ii. **egrep -i**
      Makes the pattern matching disregard the case of letters.
      A. **egrep -i 'hello'** will match a line containing "HELLO" or "Hello", as well as "hello".
      iii. **egrep -c**
      Prints the number of lines that match the search criteria.
5. Some advice on regular expressions:

(a) When constructing regular expressions, be careful to consider substrings that may match. For example, 'to' matches the word "to", but also "too", "tool", etc. 'atom' matches "neuroanatomy". If you're looking for the 718th atom of a file, '718' will get you that, but will also match the coordinate value "2.07184". It is helpful to think of what text is and isn't allowed to come before and after the pattern you are searching for. Often, this leads to seemingly semantic but often very important debates about things like "what is the definition of a word?" Recall the behavior of the **`wc -w`** command.

(b) **`egrep -v`** is an important part of testing your regular expressions. It is just as important to know about the false negatives as it is to know about the false positives.

(c) If you have the luxury, experiment on different data sets. What works on one PDB file may not work on others because the typical content may be a bit different.

(d) Learn from others. Looking at someone else's regular expressions can be intimidating at first, but it is worth the effort to break them down. You can learn a lot of useful tricks and idioms this way.

(e) You don't always have to pack everything into a single regular expression. Commands of the form **`cat x | egrep`** *<pat1>* **`| egrep`** *<pat2>* **`| egrep -v`** *<pat3>* **`> y`** can often be easier to develop (and understand) than a single RE that does the same thing.

6. Not all UNIX flavors support the same RE options. Consult the man pages for specifics. The information we give here applies to most modern UNIX systems. A useful website for intuitive creation and understanding of regular expressions can be found at: `https://www.regular-expressions.info/`

11 Exercise

1. Read (or skim) the man pages for **`egrep`**.
   (a) Try to read the whole section entitled "Regular Expressions".
   (b) What does the **`?`** operator do? Note that this operator is not available on all UNIX flavors.
   (c) Can you decipher the differences among **`grep`**, **`egrep`**, and **`fgrep`**.
2. Consider this regular expression:
   (a) **`[0-9]{5}(-[0-9]{4})?`**
   (b) Which of the above commands will it work with?
   (c) What kind of pattern does it match?
3. Write a command to extract all of the lines corresponding to the CA atoms of PRO and ALA residues in the 1DZI.pdb file.
   (a) **`egrep '^ATOM' 1DZI.pdb | egrep ' CA ' | egrep ' (PRO|ALA) '`**
4. Develop a regular expression that identifies United States Social Security numbers.
   (a) There is no sample file for this exercise. You need to make one yourself that has correctly and incorrectly formatted entries.
   (b) One of the challenges of testing regular expressions (and UNIX commands in general) is trying to come up with counter-examples that 'break' your solution.
   (c) Swap your solution with your neighbor's. Can you break your neighbor's solution? Was he able to find an example that did not work with yours? If so, refine your solution until neither of you are able to find a counter-example.
5. Same as above, but develop a solution to recognize North American telephone numbers. Consider the following possibilities.
   (a) 555-1234
   (b) 212-555-1234
   (c) (212) 555-1234
6. CHALLENGE (optional): Same as above, but develop a solution to recognize lines in a file containing a well formed number on a line by itself.

(a) Consider what the definition of a well-formed number is? Which of these formats do you want to support?
   i. Numbers written with commas and periods
      A. 186,282.3
   ii. Negative numbers:
      A. with a leading minus sign: e.g., -5
      B. accounting style: e.g., (34,231.45)
   iii. Exponential notation:
      A. FORTRAN style: e.g., 6.02E23

(b) HINT: the **`-f`** option might be useful.

7. CHALLENGE 2 (optional and especially geeky): Solve the MIT regex crossword (`https://www.mit.edu/activities/puzzle/2013/coinheist.com/rubik/a_regular_crossword/grid.pdf`)
   (a) Practice on these regex crosswords first: `https://regexcrossword.com/`

12 Miscellaneous (`date, cal, w, top`)

12.1 Lecture

1. Now it is time for some easy stuff. Let's consider a couple of miscellaneous commands that can be useful and don't involve difficult concepts like REs.
2. The **`date`** command prints the current date. There are lots of formatting options; see the man page.
3. The **`cal`** command prints calendars.
   (a) **`cal`** by itself prints a calendar for the current month.
   (b) **`cal 2020`** prints a calendar for this year.
   (c) **`cal 12 2020`** prints a calendar for December of this year.
   (d) **`cal 12 20`** prints a calendar for December of the year 20.
4. To see who else is logged into a computer, how busy it is, and how long it has been running, type the **`w`** command.
5. To see what a computer is working the hardest at, use the **`top`** command.
   (a) The display will be updated every couple of seconds.
   (b) HINT: press the **`q`** key to get out of top.

12.2 Exercise

1. Experiment with the cal command.
   (a) Try printing calendars for February of the following years: 1999, 2000, 2004, 2008, 2100, 2400, 3000, 3200, 3400. Do the leap years look right?
   (b) Try printing a calendar for the year 1752. What's up with September?
      i. Check out: `https://en.wikipedia.org/wiki/Calendar_(New_Style)_Act_1750`
   (c) MORAL: UNIX is a mature operating system. Most of the little details have been worked out.
2. Try the **`w`** command.
   (a) How long has it been since our server has been shut down?
   (b) How many people are logged in?
   (c) MORAL: UNIX is robust.
3. Try the **`top`** command.
   (a) Press `h` to read a summary of what top can do.

13 Surfing Efficiently in `vi`

13.1 Lecture

1. You can scroll around your document one screen at a time, either forwards or backwards, using CTRL-F (forward) or CTRL-B (backward).
2. You can also go directly to the first character of a particular line by typing a colon followed by the line number you want to go to.
   (a) **`:23`** brings you to line 23 of your file.
   (b) **`:1`** brings your cursor to line 1 position 1 of your document.
   (c) Pressing **`G`** brings you to the last line in the file.
3. You can also use relative expressions like the following:
   (a) **`:+9`** moves you 9 lines down.
   (b) **`:-6`** moves you 6 lines up.
4. To move to the beginning or end of the line the cursor is on, we can use **`0`** or the `$` (respectively).
   (a) **The UNIX way: `$` is a common UNIX idiom, often used to represent the end of whatever it is you happen to be dealing with lines, documents, etc.**
5. You can move around in word chunks by using the **`w`** and **`W`** keys. The lowercase **`w`** will treat all punctuation marks as separate words, while the **`W`** key treats them as if they are part of the word they adjoin. You can also move backwards by word using the **`b`** and **`B`** keys.
   (a) **`w`** treats **`it's`** as three words: 1) `it` 2) `'` and 3) `s`.
   (b) **`W`** treats **`it's`** as a single word.
6. Finally, to jump to a particular word, we can use the \ and **`?`** operators. These behave much like the find command does in a word processor. \ followed by the word you are looking for (or part of a word) brings your cursor over the first letter of the instance of that word in your document. **`?`** does the same thing, but looks backwards from your current cursor position. In fact, the 'word' here can be a regular expression; this is a very powerful way of finding just about anything in a file.

13.2 Exercise

1. Let's try moving around in the "Stairway to Heaven" lyrics file (~`unixinst/stairway.txt`).
   (a) Try moving to the beginning and end of the first line.
      i. Use **`0`** and `$`.
   (b) Now scroll down a page.
      i. Use CTRL-F.
      ii. What line number are you on now? This number may differ from your neighbor's, depending on how big your terminal window is.
   (c) Move forward 25 lines. In really big files, it can be useful to move around in chunks of 100 lines or so, rather than page by page.
      i. Use **`:+25`**.
   (d) Go to the line which contains the word 'Queen'.
      i. **`/Queen`**.
      ii. Note that this find function is case-sensitive.
   (e) Got to the beginning of the line, and then move forward 5 words.
      i. Which word are you on? If you used **`w`**, you ended up on the word 'spring'. If you used **`W`**, you ended up on the word 'for'.
      ii. Shortcut: instead of pressing the **`W`** key 5 times, you can press **`5`** followed by **`W`**. This is interpreted as "do the following thing 5 times; move forward a word".
   (f) You should now be able to get to any position in the file in just a few keystrokes (UNIX is all about svng kystrks, remember?)

14 Please, No, Not More `vi`

14.1 Lecture

1. Adding text:
   (a) Say we wanted to put the title of the song and the artist at the top of the file. We already know one way of doing that. Go to the top of the file ...
      i. **`:1`**
   (b) Next, press **`i`** to enter input mode, and then type `Stairway to Heaven -- Led Zeppelin` followed by a ⟨Return⟩ or two. Then press ⟨Escape⟩ and you're back in surfing mode. Pressing **`I`**, no matter where in the line you are, will bring the cursor to the beginning of that line, ready and waiting for you to start typing whatever it is you want to insert at that point. **`A`** does the same thing at the end of the current line.
   (c) Another way of doing this is to go to the top of the file, and press **`O`**. This is another keystroke which allows you to enter input mode. It stands for "Open a new line above the current line." Then we type the same thing, but now it is a lot more obvious that we are writing on a new line. **`o`** opens a new line below the current line. Remember, whenever you are in input mode, press ⟨Escape⟩ to get back to surfing mode.
2. Changing and deleting text:
   (a) We already know about using **`x`** to delete a single character. But to change one character, we need 4 keystrokes (count them). That seems a little excessive. The **`r`** key cuts your keystroke count in half by allowing you to replace the character that your cursor is on with whatever the next character you type is. But changing documents one character at a time can be a little tedious. Surely there must be a smarter way of doing that?
   (b) Indeed there is. The **`R`** key replaces the text under your cursor, moves onto the next character and continues to overwrite text until you press ⟨Escape⟩.
   (c) There is another command, **`cw`**, that allows you to change the word you are in. This is like highlighting a word in a word processor and replacing its text with something else. Note, however, that **`cw`** only changes text from the current cursor position to the end of the word. If you want to change the whole word, your cursor must be on the first letter.
   (d) How about deleting text? The **`d`** key is used for this. Pressing **`dw`** will delete from the cursor to the end of the word. **`dd`** will delete the whole line. **`d$`** deletes from the cursor position to the end of the line.
      i. To repeat a command in surfing mode, use the **`.`** key. This again saves keystrokes. To delete 10 lines, we can press **`dd`** and then **`.`** 9 times.
   (e) What if we want to change lots of words all at the same time? Say we wanted to change all the occurrences of 'oh' in the lyrics to 'ah'. Do we have to do them all one by one? No ...
   (f) A substitute command is available. It is a bit like doing a find and replace in a word processor. This is another example of the command mode. To use this, we need to give the command three pieces of information. First, it needs to know on which lines we want the substitution(s) to occur. Second, it needs to know the text we want to replace; finally, it needs the text we want to replace it with.
      i. **`:1,10s/oh/ah/`**
      ii. This says that we want to substitute the word 'oh' with the word 'ah' in lines 1 through 10. The **`s`** indicates that we are using the substitute command.
      iii. We can do this for the whole file by using **`:1,$`** before the **`s`**.
      iv. You can also use **`:%s`** for substitution through a whole file.
      v. Just as in moving around, we can use relative line numbers to indicate which lines we want our command to be executed on.
         A. **`:.,+8`** means from the current line (**`.`**) through the next 8 lines.

vi. However, if you try this, you'll see that only the first 'oh' on any given line is being substituted. To substitute all occurrences on any given line, we have to tack a **g** onto the end of the command.
   A. **:1,$s/oh/ah/g**

vii. Be careful with this! If there are any words which contain the letters 'oh', they will also be substituted. How can we type the command such that only whole words will be substituted?
   A. HINT: The text that you want to be substituted is a regular expression (these REs just keep popping up everywhere).
   B. Consider another example. **:1,$s/^[A-Z]//** will delete all the uppercase letters that appear at the beginning of a line.
   C. What do you think **:1,$s/to//g** will do?

viii. Substitutions can be dangerous if you don't construct your REs carefully. To be safe, you can append **c** to the end of substitute commands. This causes **vi** to ask you to confirm each substitution. To accept a substitution, type **y** and press ⟨Return⟩. This can get tedious, though.
   A. Note: When using REs in the substitute command, an ambiguous regular expression will find the longest possible match.

(g) We can also use the command mode to delete lines in bulk.
   i. Can you guess what **:1,$d** does?

(h) **vi** puts whatever you last deleted into its buffer. You can then paste the buffer contents anywhere else in the file. Move to the line above the position you want to paste your text, and press **p**. To paste to the line above your cursor position, use **P**.
   i. If you want to copy some text and paste it somewhere else in the file, you can use the yank command (the **y** key).
      A. What will the following set of commands do?
      :1,6y
      G
      p

3. Tricks with substitute
   (a) Sometimes you'll have to edit a file that was created or edited on a PC. In such cases, you'll notice that each line ends with a CTRL-M, which is a little annoying to look at. This is because the carriage return symbol for PCs and UNIX systems is different. Luckily, there is a way to get rid of them. To do this, we need to know about CTRL-V. Typing this in the substitute command tells **vi** that a special, non-alphanumeric, character is coming up next.
      i. **:1,$s/CTRL-VCTRL-M//** removes all those nasty CTRL-Ms from the end of each line.
      ii. See also the **dos2unix** command.
   (b) You can also use CTRL-V to substitute or insert tabs.

14.2 Exercise

1. Change the "Stairway to Heaven" lyrics file so that every 'lady' is capitalized.
2. On every "Woe oh oh oh oh oh" line of the chorus, change all the spaces to tabs.
3. Jump to the end of the file. Then delete the last six lines.
4. Insert a copy of the chorus ("Woe oh oh oh oh oh; And she's buying a stairway to heaven") after each stanza that doesn't already have one. DO NOT TYPE the chorus over and over, or you may go mad.

15 Non-interactive Editing (**sed**)

15.1 Lecture

1. How about if we know what we want to substitute in a file? Do we always have to use **vi**? No there is a command line version of **vi** called **sed**. The substitute command in **sed** is essentially the same as in **vi**; in particular, this includes the use of regular expressions. **sed** is especially useful when pipelining a series of commands.
   (a) **cat stairway.txt | sed 's/lady/Lady/'** will capitalize all occurrences of lady.
   (b) How can we check that this worked?
2. **sed** will automatically try to make conversions in the whole file. You can, however, specify what parts of the file you want **sed** to manipulate.
   (a) **cat stairway.txt | sed '/I/s/t.*e/garbage/g'** will convert all occurrences of the longest possible match of `t<some number of characters>e` to 'garbage' only on those lines that contain an uppercase I.
   (b) This turns the song into garbage.

16 Running Programs (environment variables, `alias`, `which`, background jobs)

16.1 Lecture

1. So far we have looked at using commands that are common to most UNIX systems. In this section, we'll consider how to run a program that is not part of the operating system, but was installed at a later date. We'll use the popular BLAST sequence analysis program as an example.
   (a) To run a BLAST job, type the commands below. The BLAST analysis will take several minutes to run.
      i. **`export PATH=~unixinst/blast/ncbi-blast-2.9.0+/bin:$PATH`**
      ii. **`export BLASTDB=~unixinst/blast/blastdb`**
      iii. **`blastp -db swissprot -evalue 1e-20`**
      `< ~unixinst/blast/app_prots.fasta > app_prots.bout`
   (b) You can run a command in the background by adding an ampersand to the end of the command.
      i. **`blastp -db swissprot -evalue 1e-20`**
      `< ~unixinst/blast/app_prots.fasta > app_prots.bout &`
      ii. While this is running, you can do other work. UNIX will tell you when the job is done. This only applies to the current window, and the check is made only when you press ⟨Return⟩.
      iii. To see jobs running in your current window, use the **`jobs`** command.
   (c) You can move running commands between the background and foreground with the techniques below. Note that they only work in the current window.
      i. CTRL-Z suspends the foreground job.
      ii. **`bg`** moves the (suspended) foreground job to the background.
      iii. **`jobs`** shows what commands are running in the background.
      iv. **`fg`** ⟨*job_id*⟩ moves the (running) background job to the foreground.
         A. When referring to background jobs in a ⟨*job_id*⟩, prefix the job number with a percent sign. For example, **`fg %2`**
      v. **`kill`** ⟨*job_id*⟩ terminates the specified background job.
   (d) If you have jobs running in the background and you log out, those jobs will continue to run.
      i. Any output that they would have sent to the screen is lost, so be sure to redirect both *stdout* and *stderr* to a file.
      ii. This technique is commonly used to submit jobs that take hours or months to run.
2. For commands that you run frequently, you can define an alias.
   (a) **`alias ls='ls sF'`**
   (b) **`alias seqs='cd ~unixinst/sequences'`**
   (c) **`alias myblast='export BLASTDB=~unixinst/blast/blastdb;`**
   `blastp -db swissprot -evalue 1e-20 <'`
3. To make your aliases and environment variable changes permanent, add the commands to your `.bash_profile` file. Use **`vi`** to do this.
   (a) The `.bash_profile` script file is run every time you log into a bash shell. Changes you make won't take effect until you log in again.
   (b) This script file contains important setup commands for your account. It may be a good idea to make a backup copy of it before you make changes to it.
4. Almost every command you type is a program that gets run.
   (a) Under normal circumstances, you have to type the full pathname to the program.
   (b) But . . . UNIX will look for a command you type in all directories that are "on your PATH".
      i. **`echo $PATH`**
      ii. The PATH is searched in order.
      iii. Note also the `$` before the variable name.

(c) Use the **`which`** command to see where a command that you run is actually stored on the system.
   i. **`which blastp`**
   ii. Note what happens if you have defined an alias.
      A. **`which ls`**
   iii. To add a directory permanently to your PATH, add a line like this to your `.bash_profile` file.
      A. **`export PATH=~unixinst/blast/ncbi-blast-2.9.0+/bin:~unixinst/bin:$PATH`**
      B. Now any executables in the instructor's `bin` directory, as well as the BLAST executables, are available to you, without having to type the full pathname.

16.2 Exercise

1. Consult the man page for the **`tail`** command to learn about the **`-f`** option.
2. Run the BLAST job as demonstrated above. Try suspending the job and running it in the background.
3. Use **`tail -f`** to monitor the progress of the analysis while your BLAST job is running in the background.
   (a) HINT: Use CTRL-C to get out of the **`tail -f`** command. It will not exit on its own, even when the BLAST job itself has finished.
4. Add the instructor's `bin` directory to your PATH.
   (a) Now run the **`progress`** command.
   (b) The **`progress`** command is a shell script. Take a look at its contents.
   (c) You will learn how to write shell scripts in the last lecture of this course.

17 Manipulating Data (`cut, csplit, sort, uniq`)

17.1 Lecture

1. The **`cut`** command is used to extract selected columns or fields from a file.
   (a) When processing by field, lines that do not contain any field delimiters will be passed though to *stdout* untouched. This behavior can be overridden by using the **`s`** option, which suppresses those lines.
   (b) Example: Extract just the x, y, and z positions of all of the CA atoms in `1DZI.pdb` into a file called `1dzi_CA.coor`.
      i. **`cat 1DZI.pdb | egrep '^ATOM' | cut -c 14-16,31-38,39-46,47-54 | \`**
      `egrep '^CA ' | cut -c 4-11,12-19,20-27 > 1dzi_CA.coor`
2. The **`csplit`** command splits up a file into subfiles. You can specify the patterns that divide up the files.
   (a) Example: Divide a fasta file containing multiple sequences into individual files
      i. **`csplit -skf seq csplit/app_prots_5.fasta '%^>%' '/^>/' '{3}'`**
3. The **`sort`** command sorts files.
   (a) You can specify which field to sort by using the **`-k`** option.
      i. **`sort`** assumes that fields are separated by whitespace. You can change the field delimiter with the **`-t`** option.
   (b) To sort in reverse order, use the **`-r`** option.
   (c) To sort in numerical order, use the **`-n`** option.
   (d) Example: sort all the files in the instructor's sequences directory by size:
      i. **`cd ~unixinst/sequences ; ls -s | egrep -v '^total' | sort -n`**
4. Another interesting command to know about is the **`uniq`** command. We won't cover it here, but you should know it is available. It helps you find unique lines of fields. You might use the **`uniq`** command, for example, to list out the different kinds of keywords that a PDB file can begin with, or to list the types of amino acids that appear in such a protein structure file.

17.2 Exercise

1. What does this do?
   (a) **`cd ~unixinst/sequences ; wc -c * | egrep -v 'total' | \`**
   `sort -rn | head -n 1 | cut -d' ' -f 2`
2. Read the man pages for the **`uniq`** command.

18 Compressing and Archiving Files (**tar, gzip, gunzip**)

18.1 Lecture

1. In this short section, we will cover three commands for managing files. The first of these is the **tar** command.
   (a) The **tar** command lets you pack a bunch of files into one large archive file. **tar** stands for tape archive. Its original use was to copy files to and from magnetic tapes. This can be handy for organizational purposes as well. If you've made a directory for a project that contains many files and subdirectories, and want to send that project information to a collaborator in another institution, you can create a single file `.tar` archive of your project directory and email it to him.
   (b) **tar** is also useful for backups. If you're about to run a command or script that could potentially damage a whole directory of files (perhaps if you are going to experiment with regular expressions), it may be a good idea to "tar up" that directory before you experiment.
   (c) tar is still used for tape backups!
   (d) **tar -cvf MyArchive.tar ~unixst01/*** will create a new archive file of the first student's home directory, and all the not-hidden files in it.
      i. The **-c** option tells **tar** to create a new archive.
      ii. The **-v** option tells **tar** to be verbose. This causes **tar** to print messages about its progress.
         A. **The UNIX way: This is another UNIX idiom; many commands use a -v option to request verbose operation.**
      iii. The **-f** option tells **tar** to use the following argument as the archive file name.
         A. Generally, command options can be specified in any order; however, in this case **-f** should be the last option on the list because it expects the filename to follow.
      iv. It is probably not a good idea to "tar up" your own directory and at the same time store the archive you're making in that directory. This is telling the archive to archive itself. But you can ask your neighbor to **tar** up your directory, then **cp** that file into your own directory.
         A. Another option would be to create the archive in the `/tmp` directory on your system.
      v. Note that `tar` archives are traditionally named with a `.tar` suffix.
   (e) **tar -tf MyArchive.tar** shows the contents of the archive.
      i. The **-t** option tells tar to list a table of contents of the archive.
   (f) **tar -xvf MyArchive.tar** unpacks, or "untars", the archive.
      i. Be careful when untarring an archive, as your current (presumably more recent) files can be overwritten if a filename in the archive happens to match a filename in your directory. It is usually best to create an empty directory, **cd** into it, and untar your archive there. Then you can move files and directories around as needed with the **mv** and **cp** commands.
      ii. The tarred files will have the same relative structure as when they were created. It's a good idea to think about where you want to be relative to the files you want to archive. Compare:
         A. **cd ~/work; tar -cvf MyArchive.tar ~unixst01/***
         B. **cd ~unixst01/..; tar -cvf ~/work/MyArchive.tar unixst01/***
      iii. It is possible to extract a single file from a tar archive without having to unpack the whole thing. You guessed it: see the man page!
2. Finally, we consider a complementary pair of commands that compress and uncompress files. Although disk space on computers keeps getting bigger and less expensive, it is never infinite. At some point, you'll probably need to squeeze some extra space out of your account.
   (a) One option is to **tar** up dormant project directories and ask your system administrator to move the archive to tape.
   (b) Another option is to compress files that you won't be using for a while.
      i. **gzip app_prots.bout** will compress the output of the BLAST job you ran earlier.

A. The compressed file is named `app_prots.bout.gz`.
B. The original file is removed.
C. How much smaller did it get?

ii. **`gunzip app_prots.bout.gz`** will uncompress the file.
A. **`gzip`**'s compression is lossless. If you compress and uncompress a file, you get back exactly what you had.
B. Some files compress better than others.

18.2 Exercise

1. **`tar`** up your neighbor's home directory. How large is the resultant `tar` file?
2. Compress that `tar` file. How big is it now?
   (a) `.tar.gz` files are commonly used in UNIX. Much of the software and data you download from the web will be in `.tar.gz` format. These are also commonly suffixed at `.tgz`, and are referred to as 'tarballs'.
   (b) As a general rule of thumb, you can expect about 50% compression from a sufficiently large sample of "normal" files. Your mileage, of course, may vary.
3. Get a copy of the `.tar.gz` file of your directory from your neighbor.
4. Convince yourself that your neighbor did a good job of backing up your home directory by listing the contents of the archive.
5. Try asking for a verbose table of contents.
6. OPTIONAL: "Accidentally" delete a (not-so-important) file from your directory. Have your neighbor recover it for you using the archive he made.

