Bash-UNIX_CarpentriesBootcamp

The Carpentries, edited by Vanessa Guerra and Serena Caplins

3/25/2021

Working With Files and Directories

Before we start, make sure to clone or update the github folder MarineGenomics in the user directory

```
$ cd /home/margeno/
$ git clone https://github.com/BayLab/MarineGenomics
```

Creating directories

We now know how to explore files and directories, but how do we create them in the first place?

Step one: see where we are and what we already have

Let's open the folder Week2 inside of /home/margeno/MarineGenomics/data/Week2 and see where what it contains with `ls -F` :

Bash

```
$ cd /home/margeno/MarineGenomics/data/Week2/data-shell
```

```
$ pwd
```

Output

```
/home/margeno/MarineGenomics/data/Week2/data-shell
```

Bash

```
$ ls -F
```

Output

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

Create a directory

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

Bash

```
$ mkdir thesis
```

As you might guess from its name, `mkdir` means 'make directory'. Since `thesis` is a relative path (i.e., does not have a leading slash, like `/what/ever/thesis`), the new directory is created in the current working directory:

Bash

```
$ ls -F
```

Output

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

Since we've just created the `thesis` directory, there's nothing in it yet:

Bash

```
$ ls -F thesis
```

Note that `mkdir` is not limited to creating single directories one at a time. The `-p` option allows `mkdir` to create a directory with any number of nested subdirectories in a single operation:

Bash

```
$ mkdir -p thesis/chapter_1/section_1/subsection_1
```

The `-R` option to the `ls` command will list all nested subdirectories within a directory. Let's use `ls -FR` to recursively list the new directory hierarchy we just created beneath the `thesis` directory:

Bash

```
$ ls -FR thesis
chapter_1/

thesis/chapter_1:
section_1/

thesis/chapter_1/section_1:
subsection_1/

thesis/chapter_1/section_1/subsection_1:
```

Two ways of doing the same thing

Using the shell to create a directory is no different than using a file explorer. If you open the current directory using your operating system's graphical file explorer, the `thesis` directory will appear there too. While the shell and the file explorer are two different ways of interacting with the files, the files and directories themselves are the same.

Good names for files and directories

Complicated names of files and directories can make your life painful when working on the command line. Here we provide a few useful tips for the names of your files.

1. Don't use spaces.

   Spaces can make a name more meaningful, but since spaces are used to separate arguments on the command line it is better to avoid them in names of files and directories. You can use `-` or `_` instead (e.g. `north-pacific-gyre/` rather than `north pacific gyre/`).
2. Don't begin the name with `-` (dash).

   Commands treat names starting with `-` as options.
3. Stick with letters, numbers, `.` (period or 'full stop'), `-` (dash) and `_` (underscore).

   Many other characters have special meanings on the command line. We will learn about some of these during this lesson. There are special characters that can cause your command to not work as expected and can even result in data loss.

If you need to refer to names of files or directories that have spaces or other special characters, you should surround the name in quotes (`""`).

Create a text file

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

Bash

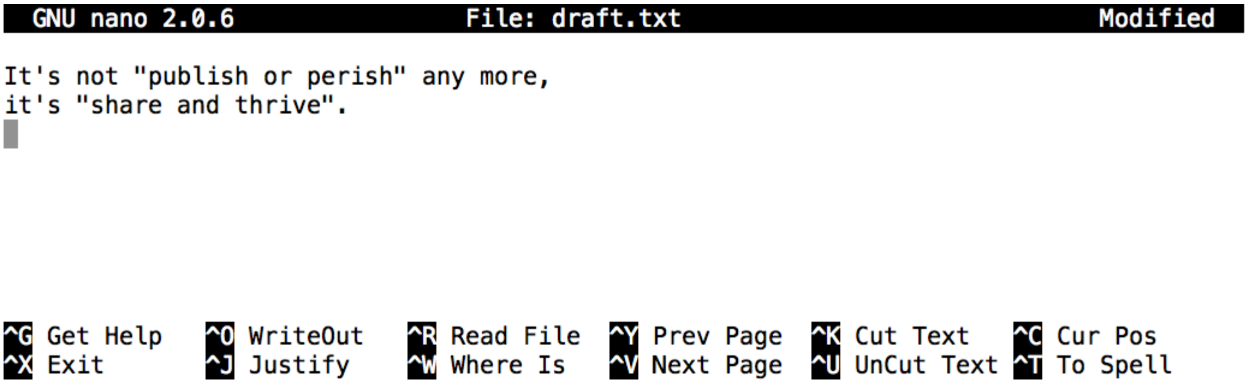
```
$ cd thesis
$ nano draft.txt
```

Which Editor?

When we say, '`nano` is a text editor' we really do mean 'text': it can only work with plain character data, not tables, images, or any other human-friendly media. We use it in examples because it is one of the least complex text editors. However, because of this trait, it may not be powerful enough or flexible enough for the work you need to do after this workshop. On Unix systems (such as Linux and macOS), many programmers use Emacs or Vim (both of which require more time to learn), or a graphical editor such as Gedit. On Windows, you may wish to use Notepad++. Windows also has a built-in editor called `notepad` that can be run from the command line in the same way as `nano` for the purposes of this lesson.

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

Let's type in a few lines of text. Once we're happy with our text, we can press Ctrl+O (press the Ctrl or Control key and, while holding it down, press the O key) to write our data to disk (we'll be asked what file we want to save this to: press Return to accept the suggested default of `draft.txt`).



```
  GNU nano 2.0.6                File: draft.txt                         Modified

It's not "publish or perish" any more,
it's "share and thrive".
```

Once our file is saved, we can use Ctrl+X to quit the editor and return to the shell.

Control, Ctrl, or ^ Key

The Control key is also called the 'Ctrl' key. There are various ways in which using the Control key may be described. For example, you may see an instruction to press the Control key and, while holding it down, press the X key, described as any of:

- `Control-X`
- `Control+X`
- `Ctrl-X`
- `Ctrl+X`
- `^X`
- `C-x`

In nano, along the bottom of the screen you'll see `^G Get Help ^O WriteOut`. This means that you can use `Control-G` to get help and `Control-O` to save your file.

`nano` doesn't leave any output on the screen after it exits, but `ls` now shows that we have created a file called `draft.txt`:

Bash

```
$ ls
```

Output

```
draft.txt
```

Creating Files a Different Way

We have seen how to create text files using the `nano` editor. Now, try the following command:

Bash

```
$ touch my_file.txt
```

1. What did the `touch` command do? When you look at your current directory using the GUI file explorer, does the file show up?
2. Use `ls -l` to inspect the files. How large is `my_file.txt`?
3. When might you want to create a file this way?

Solution

Solution

1. The `touch` command generates a new file called `my_file.txt` in your current directory. You can observe this newly generated file by typing `ls` at the command line prompt. `my_file.txt` can also be viewed in your GUI file explorer.
2. When you inspect the file with `ls -l`, note that the size of `my_file.txt` is 0 bytes. In other words, it contains no data. If you open `my_file.txt` using your text editor it is blank.
3. Some programs do not generate output files themselves, but instead require that empty files have already been generated. When the program is run, it searches for an existing file to populate with its output. The touch command allows you to efficiently generate a blank text file to be used by such programs.

What's In A Name?

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

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

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

Moving files and directories

Returning to the `data-shell` directory,

Bash

```
cd /home/margeno/MarineGenomics/data/Week2/data-shell
```

In our `thesis` directory we have a file `draft.txt` which isn't a particularly informative name, so let's change the file's name using `mv`, which is short for 'move':

Bash

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

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

Bash

```
$ ls thesis
```

Output

```
quotes.txt
```

One has to be careful when specifying the target file name, since `mv` will silently overwrite any existing file with the same name, which could lead to data loss. An additional option, `mv -i` (or `mv --interactive`), can be used to make `mv` ask you for confirmation before overwriting.

Note that `mv` also works on directories.

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

Bash

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

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

Bash

```
$ ls thesis
```

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

Bash

```
$ ls quotes.txt
```

Output

```
quotes.txt
```

Moving Files to a new folder

After running the following commands, Jamie realizes that she put the files `sucrose.dat` and `maltose.dat` into the wrong folder. The files should have been placed in the `raw` folder.

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

Fill in the blanks to move these files to the `raw/` folder (i.e. the one she forgot to put them in)

Bash

```
$ mv sucrose.dat maltose.dat ____/____
```

Solution

Solution

Bash

```
$ mv sucrose.dat maltose.dat ../raw
```

Recall that `..` refers to the parent directory (i.e. one above the current directory) and that `.` refers to the current directory.

Copying files and directories

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

Bash

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

Output

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

We can also copy a directory and all its contents by using the recursive option `-r`, e.g. to back up a directory:

Bash

```
$ cp -r thesis thesis_backup
```

We can check the result by listing the contents of both the `thesis` and `thesis_backup` directory:

Bash

```
$ ls thesis thesis_backup
```

Output

```
thesis:
quotations.txt

thesis_backup:
quotations.txt
```

Renaming Files

Suppose that you created a plain-text file in your current directory to contain a list of the statistical tests you will need to do to analyze your data, and named it: `statstics.txt`

After creating and saving this file you realize you misspelled the filename! You want to correct the mistake, which of the following commands could you use to do so?

1. `cp statstics.txt statistics.txt`
2. `mv statstics.txt statistics.txt`
3. `mv statstics.txt .`
4. `cp statstics.txt .`

Solution

Solution

1. No. While this would create a file with the correct name, the incorrectly named file still exists in the directory and would need to be deleted.
2. Yes, this would work to rename the file.
3. No, the period(.) indicates where to move the file, but does not provide a new file name; identical file names cannot be created.
4. No, the period(.) indicates where to copy the file, but does not provide a new file name; identical file names cannot be created.

Moving and Copying

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

Bash

```
$ pwd
```

Output

```
/Users/jamie/data
```

Bash

```
$ ls
```

OUtput

```
proteins.dat
```

Bash

```
$ mkdir recombined
$ mv proteins.dat recombined/
$ cp recombined/proteins.dat ../proteins-saved.dat
$ ls
```

1. `proteins-saved.dat recombined`
2. `recombined`
3. `proteins.dat recombined`
4. `proteins-saved.dat`

Solution

Solution

We start in the `/Users/jamie/data` directory, and create a new folder called `recombined`. The second line moves (`mv`) the file `proteins.dat` to the new folder (`recombined`). The third line makes a copy of the file we just moved. The tricky part here is where the file was copied to. Recall that `..` means 'go up a level', so the copied file is now in `/Users/jamie`. Notice that `..` is interpreted with respect to the current working directory, **not** with respect to the location of the file being copied. So, the only thing that will show using ls (in `/Users/jamie/data`) is the recombined folder.

1. No, see explanation above. `proteins-saved.dat` is located at `/Users/jamie`
2. Yes
3. No, see explanation above. `proteins.dat` is located at `/Users/jamie/data/recombined`
4. No, see explanation above. `proteins-saved.dat` is located at `/Users/jamie`

Removing files and directories

Returning to the `MarineGenomics` directory, let's tidy up this directory by removing the `quotes.txt` file we created. The Unix command we'll use for this is `rm` (short for 'remove'):

```
$ rm quotes.txt
```

We can confirm the file has gone using `ls`:

```
$ ls quotes.txt
```

```
ls: cannot access 'quotes.txt': No such file or directory
```

Deleting Is Forever

The Unix shell doesn't have a trash bin that we can recover deleted files from (though most graphical interfaces to Unix do). Instead, when we delete files, they are unlinked from the file system so that their storage space on disk can be recycled. Tools for finding and recovering deleted files do exist, but there's no guarantee they'll work in any particular situation, since the computer may recycle the file's disk space right away.

Using `rm` Safely

What happens when we execute `rm -i thesis_backup/quotations.txt`? Why would we want this protection when using `rm`?

Solution

Solution

```
$ rm: remove regular file 'thesis_backup/quotations.txt'? y
```

The `-i` option will prompt before (every) removal (use Y to confirm deletion or N to keep the file). The Unix shell doesn't have a trash bin, so all the files removed will disappear forever. By using the `-i` option, we have the chance to check that we are deleting only the files that we want to remove.

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

```
$ rm thesis
```

Error

```
rm: cannot remove ‘thesis’: Is a directory
```

This happens because `rm` by default only works on files, not directories.

`rm` can remove a directory *and all its contents* if we use the recursive option `-r`, and it will do so *without any confirmation prompts*:

```
$ rm -r thesis
```

Given that there is no way to retrieve files deleted using the shell, `rm -r` *should be used with great caution* (you might consider adding the interactive option `rm -r -i`).

Operations with multiple files and directories

Oftentimes one needs to copy or move several files at once. This can be done by providing a list of individual filenames, or specifying a naming pattern using wildcards.

Copy with Multiple Filenames

For this exercise, you can test the commands in the `data-shell/data` directory.

In the example below, what does `cp` do when given several filenames and a directory name?

```
$ mkdir backup
$ cp amino-acids.txt animals.txt backup/
```

In the example below, what does `cp` do when given three or more file names?

```
$ ls -F
```

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

```
$ cp amino-acids.txt animals.txt morse.txt
```

Solution

Solution

If given more than one file name followed by a directory name (i.e. the destination directory must be the last argument), `cp` copies the files to the named directory.

If given three file names, `cp` throws an error such as the one below, because it is expecting a directory name as the last argument.

Output

```
cp: target ‘morse.txt’ is not a directory
```

Wildcards

`*` is a **wildcard**, which matches zero or more characters. Let's consider the `data-shell/molecules` directory: `*.pdb` matches `ethane.pdb`, `propane.pdb`, and every file that ends with '.pdb'. On the other hand, `p*.pdb` only matches `pentane.pdb` and `propane.pdb`, because the 'p' at the front only matches filenames that begin with the letter 'p'.

`?` is also a wildcard, but it matches exactly one character. So `?ethane.pdb` would match `methane.pdb` whereas `*ethane.pdb` matches both `ethane.pdb`, and `methane.pdb`.

Wildcards can be used in combination with each other e.g. `???ane.pdb` matches three characters followed by `ane.pdb`, giving `cubane.pdb  ethane.pdb  octane.pdb`.

When the shell sees a wildcard, it expands the wildcard to create a list of matching filenames *before* running the command that was asked for. As an exception, if a wildcard expression does not match any file, Bash will pass the expression as an argument to the command as it is. For example typing `ls *.pdf` in the `molecules` directory (which contains only files with names ending with `.pdb`) results in an error message that there is no file called `*.pdf`. However, generally commands like `wc` and `ls` see the lists of file names matching these expressions, but not the wildcards themselves. It is the shell, not the other programs, that deals with expanding wildcards, and this is another example of orthogonal design.

List filenames matching a pattern

When run in the `molecules` directory, which `ls` command(s) will produce this output?

```
ethane.pdb   methane.pdb
```

1. `ls *t*ane.pdb`
2. `ls *t?ne.*`
3. `ls *t??ne.pdb`
4. `ls ethane.*`

Solution

Solution

The solution is `3.`

`1.` shows all files whose names contain zero or more characters (`*`) followed by the letter `t`, then zero or more characters (`*`) followed by `ane.pdb`. This gives `ethane.pdb methane.pdb  octane.pdb  pentane.pdb`.

`2.` shows all files whose names start with zero or more characters (`*`) followed by the letter `t`, then a single character (`?`), then `ne.` followed by zero or more characters (`*`). This will give us `octane.pdb` and `pentane.pdb` but doesn't match anything which ends in `thane.pdb`.

`3.` fixes the problems of option 2 by matching two characters (`??`) between `t` and `ne`. This is the solution.

`4.` only shows files starting with `ethane.`.

Solution

Pipes and Filters

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

```
$ ls molecules
```

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

Let's go into that directory with `cd` and run an example command `wc cubane.pdb`:

```
$ cd molecules
$ wc cubane.pdb
```

```
20  156 1158 cubane.pdb
```

`wc` is the 'word count' command: it counts the number of lines, words, and characters in files (from left to right, in that order).

If we run the command `wc *.pdb`, the `*` in `*.pdb` matches zero or more characters, so the shell turns `*.pdb` into a list of all `.pdb` files in the current directory:

```
$ wc *.pdb
```

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

Note that `wc *.pdb` also shows the total number of all lines in the last line of the output.

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

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

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

The `-m` and `-w` options can also be used with the `wc` command, to show only the number of characters or the number of words in the files.

Why Isn't It Doing Anything?

What happens if a command is supposed to process a file, but we don't give it a filename? For example, what if we type:

```
$ wc -l
```

but don't type `*.pdb` (or anything else) after the command? Since it doesn't have any filenames, `wc` assumes it is supposed to process input given at the command prompt, so it just sits there and waits for us to give it some data interactively. From the outside, though, all we see is it sitting there: the command doesn't appear to do anything.

If you make this kind of mistake, you can escape out of this state by holding down the control key (Ctrl) and typing the letter C once and letting go of the Ctrl key. Ctrl+C

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

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

The greater than symbol, `>`, tells the shell to **redirect** the command's output to a file instead of printing it to the screen. (This is why there is no screen output: everything that `wc` would have printed has gone into the file `lengths.txt` instead.) The shell will create the file if it doesn't exist. If the file exists, it will be silently overwritten, which may lead to data loss and thus requires some caution. `ls lengths.txt` confirms that the file exists:

```
$ ls lengths.txt
```

```
lengths.txt
```

We can now send the content of `lengths.txt` to the screen using `cat lengths.txt`. The `cat` command gets its name from 'concatenate' i.e. join together, and it prints the contents of files one after another. There's only one file in this case, so `cat` just shows us what it contains:

```
$ cat lengths.txt
```

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

Output Page by Page

We'll continue to use `cat` in this lesson, for convenience and consistency, but it has the disadvantage that it always dumps the whole file onto your screen. More useful in practice is the command `less`, which you use with `less lengths.txt`. This displays a screenful of the file, and then stops. You can go forward one screenful by pressing the spacebar, or back one by pressing `b`. Press `q` to quit.

Now let's use the `sort` command to sort its contents.

What Does `sort -n` Do?

If we run `sort` on a file containing the following lines:

```
10
2
19
22
6
```

the output is:

```
10
19
2
22
6
```

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

```
2
6
10
19
22
```

Explain why `-n` has this effect.

Solution

Solution

The `-n` option specifies a numerical rather than an alphanumerical sort.

Solution

We will also use the `-n` option to specify that the sort is numerical instead of alphanumerical. This does *not* change the file; instead, it sends the sorted result to the screen:

```
$ sort -n lengths.txt
```

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

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

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

```
  9  methane.pdb
```

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

Redirecting to the same file

It's a very bad idea to try redirecting the output of a command that operates on a file to the same file. For example:

```
$ sort -n lengths.txt > lengths.txt
```

Doing something like this may give you incorrect results and/or delete the contents of `lengths.txt`.

What Does >> Mean?

We have seen the use of `>`, but there is a similar operator `>>` which works slightly differently. We'll learn about the differences between these two operators by printing some strings. We can use the `echo` command to print strings e.g.

```
$ echo The echo command prints text
```

```
The echo command prints text
```

Now test the commands below to reveal the difference between the two operators:

```
$ echo hello > testfile01.txt
```

and:

```
$ echo hello >> testfile02.txt
```

Hint: Try executing each command twice in a row and then examining the output files.

Solution

Solution

In the first example with `>`, the string 'hello' is written to `testfile01.txt`, but the file gets overwritten each time we run the command.

We see from the second example that the `>>` operator also writes 'hello' to a file (in this case`testfile02.txt`), but appends the string to the file if it already exists (i.e. when we run it for the second time).

Solution

Appending Data

We have already met the head command, which prints lines from the start of a file. tail is similar, but prints lines from the end of a file instead.

Consider the file data-shell/data/animals.txt. After these commands, select the answer that corresponds to the file animals-subset.txt:

```
$ head -n 3 animals.txt > animals-subset.txt
$ tail -n 2 animals.txt >> animals-subset.txt
```

1. The first three lines of animals.txt
2. The last two lines of animals.txt
3. The first three lines and the last two lines of animals.txt
4. The second and third lines of animals.txt

Solution

Solution

Option 3 is correct. For option 1 to be correct we would only run the head command. For option 2 to be correct we would only run the tail command. For option 4 to be correct we would have to pipe the output of head into tail -n 2 by doing head -n 3 animals.txt | tail -n 2 > animals-subset.txt

Solution

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

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

```
  9  methane.pdb
```

The vertical bar, |, between the two commands is called a **pipe**. It tells the shell that we want to use the output of the command on the left as the input to the command on the right.

Nothing prevents us from chaining pipes consecutively. That is, we can for example send the output of wc directly to sort, and then the resulting output to head. Thus we first use a pipe to send the output of wc to sort:

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

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

And now we send the output of this pipe, through another pipe, to head, so that the full pipeline becomes:

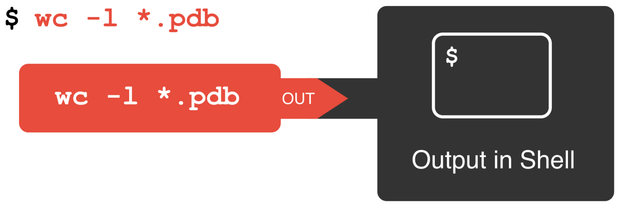
```
$ wc -l *.pdb | sort -n | head -n 1
```

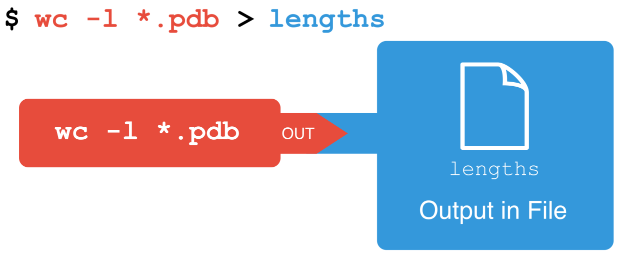
```
   9  methane.pdb
```

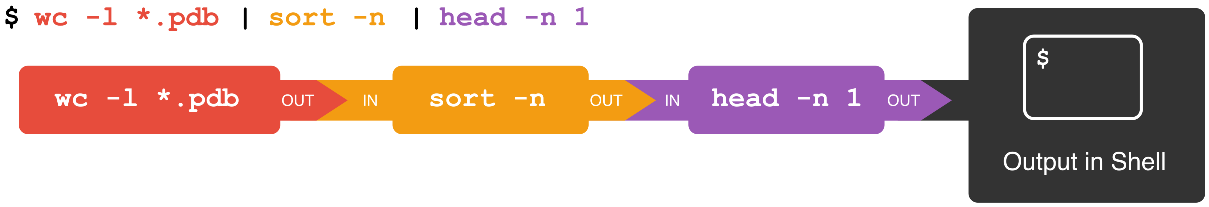
This is exactly like a mathematician nesting functions like *log(3x)* and saying ‘the log of three times *x*’. In our case, the calculation is ‘head of sort of line count of `*.pdb`’.

The redirection and pipes used in the last few commands are illustrated below:

Redirects and Pipes of different commands: “wc -l *.pdb" will direct the output to the shell. "wc -l* .pdb > lengths” will direct output to the file “lengths”. "wc -l *.pdb | sort -n | head -n 1" will build a pipeline where the output of the “wc” command is the input to the “sort” command, the output of the “sort” command is the input to the “head” command and the output of the “head” command is directed to the shell







Piping Commands Together

In our current directory, we want to find the 3 files which have the least number of lines. Which command listed below would work?

1. `wc -l * > sort -n > head -n 3`
2. `wc -l * | sort -n | head -n 1-3`
3. `wc -l * | head -n 3 | sort -n`
4. `wc -l * | sort -n | head -n 3`

Solution

Solution

Option 4 is the solution. The pipe character `|` is used to connect the output from one command to the input of another. `>` is used to redirect standard output to a file. Try it in the `data-shell/molecules` directory!

Solution

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

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

Pipe Reading Comprehension

A file called `animals.txt` (in the `data-shell/data` folder) contains the following data:

```
2012-11-05,deer
2012-11-05,rabbit
2012-11-05,raccoon
2012-11-06,rabbit
2012-11-06,deer
2012-11-06,fox
2012-11-07,rabbit
2012-11-07,bear
```

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

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

Hint: build the pipeline up one command at a time to test your understanding

Solution

Solution

The `head` command extracts the first 5 lines from `animals.txt`. Then, the last 3 lines are extracted from the previous 5 by using the `tail` command. With the `sort -r` command those 3 lines are sorted in reverse order and finally, the output is redirected to a file `final.txt`. The content of this file can be checked by executing `cat final.txt`. The file should contain the following lines:

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

Solution

Pipe Construction

For the file `animals.txt` from the previous exercise, consider the following command:

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

The `cut` command is used to remove or 'cut out' certain sections of each line in the file, and `cut` expects the lines to be separated into columns by a Tab character. A character used in this way is a called a **delimiter**. In the example above we use the `-d` option to specify the comma as our delimiter character. We have also used the `-f` option to specify that we want to extract the second field (column). This gives the following output:

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

The `uniq` command filters out adjacent matching lines in a file. How could you extend this pipeline (using `uniq` and another command) to find out what animals the file contains (without any duplicates in their names)?

Solution

Solution

```
$ cut -d , -f 2 animals.txt | sort | uniq
```

Solution

Which Pipe?

The file `animals.txt` contains 8 lines of data formatted as follows:

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

The `uniq` command has a `-c` option which gives a count of the number of times a line occurs in its input. Assuming your current directory is `data-shell/data/`, what command would you use to produce a table that shows the total count of each type of animal in the file?

1. `sort animals.txt | uniq -c`
2. `sort -t, -k2,2 animals.txt | uniq -c`
3. `cut -d, -f 2 animals.txt | uniq -c`
4. `cut -d, -f 2 animals.txt | sort | uniq -c`
5. `cut -d, -f 2 animals.txt | sort | uniq -c | wc -l`

Solution

Solution

Option 4. is the correct answer. If you have difficulty understanding why, try running the commands, or sub-sections of the pipelines (make sure you are in the `data-shell/data` directory).

Solution

Nelle's Pipeline: Checking Files

Nelle has run her samples through the assay machines and created 17 files in the `north-pacific-gyre/2012-07-03` directory described earlier. As a quick check, starting from her home directory, Nelle types:

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

The output is 18 lines that look like this:

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

Now she types this:

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

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

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

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

```
 300 NENE02040B.txt
 300 NENE02040Z.txt
 300 NENE02043A.txt
 300 NENE02043B.txt
5040 total
```

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

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

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

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

Removing Unneeded Files

Suppose you want to delete your processed data files, and only keep your raw files and processing script to save storage. The raw files end in `.dat` and the processed files end in `.txt`. Which of the following would remove all the processed data files, and *only* the processed data files?

1. `rm ?.txt`
2. `rm *.txt`
3. `rm * .txt`
4. `rm *.*`

Solution

Solution

1. This would remove `.txt` files with one-character names
2. This is correct answer
3. The shell would expand `*` to match everything in the current directory, so the command would try to remove all matched files and an additional file called `.txt`
4. The shell would expand `*.*` to match all files with any extension, so this command would delete all files

Solution

Loops

Loops are a programming construct which allow us to repeat a command or set of commands for each item in a list. As such they are key to productivity improvements through automation. Similar to wildcards and tab completion, using loops also reduces the amount of typing required (and hence reduces the number of typing mistakes).

Suppose we have several hundred genome data files named `basilisk.dat`, `minotaur.dat`, and `unicorn.dat`. For this example, we'll use the `creatures` directory which only has three example files, but the principles can be applied to many many more files at once.

The structure of these files is the same: the common name, classification, and updated date are presented on the first three lines, with DNA sequences on the following lines. Let's look at the files:

```
$ head -n 5 basilisk.dat minotaur.dat unicorn.dat
```

We would like to print out the classification for each species, which is given on the second line of each file. For each file, we would need to execute the command `head -n 2` and pipe this to `tail -n 1`. We'll use a loop to solve this problem, but first let's look at the general form of a loop:

```
for thing in list_of_things
do
    operation_using $thing    # Indentation within the loop is not required, but aids legibility
done
```

and we can apply this to our example like this:

```
$ for filename in basilisk.dat minotaur.dat unicorn.dat
> do
>     head -n 2 $filename | tail -n 1
> done
```

```
CLASSIFICATION: basiliscus vulgaris
CLASSIFICATION: bos hominus
CLASSIFICATION: equus monoceros
```

Follow the Prompt

The shell prompt changes from `$` to `>` and back again as we were typing in our loop. The second prompt, `>`, is different to remind us that we haven't finished typing a complete command yet. A semicolon, `;`, can be used to separate two commands written on a single line.

When the shell sees the keyword `for`, it knows to repeat a command (or group of commands) once for each item in a list. Each time the loop runs (called an iteration), an item in the list is assigned in sequence to the **variable**, and the commands inside the loop are executed, before moving on to the next item in the list. Inside the loop, we call for the variable's value by putting `$` in front of it. The `$` tells the shell interpreter to treat the variable as a variable name and substitute its value in its place, rather than treat it as text or an external command.

In this example, the list is three filenames: `basilisk.dat`, `minotaur.dat`, and `unicorn.dat`. Each time the loop iterates, it will assign a file name to the variable `filename` and run the `head` command. The first time through the loop, `$filename` is `basilisk.dat`. The interpreter runs the command `head` on `basilisk.dat` and pipes the first two lines to the `tail` command, which then prints the second line of `basilisk.dat`. For the second iteration, `$filename` becomes `minotaur.dat`. This time, the shell runs `head` on `minotaur.dat` and pipes the first two lines to the `tail` command, which then prints the second line of `minotaur.dat`. For the third iteration, `$filename` becomes `unicorn.dat`, so the shell runs the `head` command on that file, and `tail` on the output of that. Since the list was only three items, the shell exits the `for` loop.

Same Symbols, Different Meanings

Here we see `>` being used a shell prompt, whereas `>` is also used to redirect output. Similarly, `$` is used as a shell prompt, but, as we saw earlier, it is also used to ask the shell to get the value of a variable.

If the *shell* prints `>` or `$` then it expects you to type something, and the symbol is a prompt.

If *you* type `>` or `$` yourself, it is an instruction from you that the shell should redirect output or get the value of a variable.

When using variables it is also possible to put the names into curly braces to clearly delimit the variable name: `$filename` is equivalent to `${filename}`, but is different from `${file}name`. You may find this notation in other people's programs.

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

```
$ for x in basilisk.dat minotaur.dat unicorn.dat
> do
>     head -n 2 $x | tail -n 1
> done
```

or:

```
$ for temperature in basilisk.dat minotaur.dat unicorn.dat
> do
>     head -n 2 $temperature | tail -n 1
> done
```

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

Variables in Loops

This exercise refers to the `data-shell/molecules` directory. `ls` gives the following output:

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

What is the output of the following code?

```
$ for datafile in *.pdb
> do
>     ls *.pdb
> done
```

Now, what is the output of the following code?

```
$ for datafile in *.pdb
> do
> ls $datafile
> done
```

Why do these two loops give different outputs?

Solution

Solution

The first code block gives the same output on each iteration through the loop. Bash expands the wildcard `*.pdb` within the loop body (as well as before the loop starts) to match all files ending in `.pdb` and then lists them using `ls`. The expanded loop would look like this:

```
$ for datafile in cubane.pdb  ethane.pdb  methane.pdb  octane.pdb  pentane.pdb  propane.pdb
> do
>     ls cubane.pdb  ethane.pdb  methane.pdb  octane.pdb  pentane.pdb  propane.pdb
> done
```

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

The second code block lists a different file on each loop iteration. The value of the `datafile` variable is evaluated using `$datafile`, and then listed using `ls`.

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

Solution

Limiting Sets of Files

What would be the output of running the following loop in the `data-shell/molecules` directory?

```
$ for filename in c*
> do
>     ls $filename
> done
```

1. No files are listed.
2. All files are listed.
3. Only `cubane.pdb`, `octane.pdb` and `pentane.pdb` are listed.
4. Only `cubane.pdb` is listed.

Solution

Solution

4 is the correct answer. `*` matches zero or more characters, so any file name starting with the letter c, followed by zero or more other characters will be matched.

Solution

How would the output differ from using this command instead?

```
$ for filename in *c*
> do
>     ls $filename
> done
```

1. The same files would be listed.
2. All the files are listed this time.
3. No files are listed this time.
4. The files `cubane.pdb` and `octane.pdb` will be listed.
5. Only the file `octane.pdb` will be listed.

Solution

Solution

4 is the correct answer. `*` matches zero or more characters, so a file name with zero or more characters before a letter c and zero or more characters after the letter c will be matched.

Solution

Saving to a File in a Loop - Part One

In the `data-shell/molecules` directory, what is the effect of this loop?

```
for alkanes in *.pdb
do
    echo $alkanes
    cat $alkanes > alkanes.pdb
done
```

1. Prints `cubane.pdb`, `ethane.pdb`, `methane.pdb`, `octane.pdb`, `pentane.pdb` and `propane.pdb`, and the text from `propane.pdb` will be saved to a file called `alkanes.pdb`.
2. Prints `cubane.pdb`, `ethane.pdb`, and `methane.pdb`, and the text from all three files would be concatenated and saved to a file called `alkanes.pdb`.
3. Prints `cubane.pdb`, `ethane.pdb`, `methane.pdb`, `octane.pdb`, and `pentane.pdb`, and the text from `propane.pdb` will be saved to a file called `alkanes.pdb`.
4. None of the above.

Solution

Solution

1. The text from each file in turn gets written to the `alkanes.pdb` file. However, the file gets overwritten on each loop interation, so the final content of `alkanes.pdb` is the text from the `propane.pdb` file.

Solution

Saving to a File in a Loop - Part Two

Also in the `data-shell/molecules` directory, what would be the output of the following loop?

```
for datafile in *.pdb
do
    cat $datafile >> all.pdb
done
```

1. All of the text from `cubane.pdb`, `ethane.pdb`, `methane.pdb`, `octane.pdb`, and `pentane.pdb` would be concatenated and saved to a file called `all.pdb`.
2. The text from `ethane.pdb` will be saved to a file called `all.pdb`.
3. All of the text from `cubane.pdb`, `ethane.pdb`, `methane.pdb`, `octane.pdb`, `pentane.pdb` and `propane.pdb` would be concatenated and saved to a file called `all.pdb`.
4. All of the text from `cubane.pdb`, `ethane.pdb`, `methane.pdb`, `octane.pdb`, `pentane.pdb` and `propane.pdb` would be printed to the screen and saved to a file called `all.pdb`.

Solution

Solution

3 is the correct answer. `>>` appends to a file, rather than overwriting it with the redirected output from a command. Given the output from the `cat` command has been redirected, nothing is printed to the screen.

Solution

Let's continue with our example in the `data-shell/creatures` directory. Here's a slightly more complicated loop:

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

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

```
$ echo hello there
```

prints:

```
hello there
```

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

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

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

Spaces in Names

Spaces are used to separate the elements of the list that we are going to loop over. If one of those elements contains a space character, we need to surround it with quotes, and do the same thing to our loop variable. Suppose our data files are named:

```
red dragon.dat
purple unicorn.dat
```

To loop over these files, we would need to add double quotes like so:

```
$ for filename in "red dragon.dat" "purple unicorn.dat"
> do
>     head -n 100 "$filename" | tail -n 20
> done
```

It is simpler to avoid using spaces (or other special characters) in filenames.

The files above don't exist, so if we run the above code, the `head` command will be unable to find them, however the error message returned will show the name of the files it is expecting:

```
head: cannot open ‘red dragon.dat’ for reading: No such file or directory
head: cannot open ‘purple unicorn.dat’ for reading: No such file or directory
```

Try removing the quotes around `$filename` in the loop above to see the effect of the quote marks on spaces. Note that we get a result from the loop command for unicorn.dat when we run this code in the `creatures` directory:

```
head: cannot open ‘red’ for reading: No such file or directory
head: cannot open ‘dragon.dat’ for reading: No such file or directory
head: cannot open ‘purple’ for reading: No such file or directory
CGGTACCGAA
AAGGGTCGCG
CAAGTGTTCC
```

We would like to modify each of the files in `data-shell/creatures`, but also save a version of the original files, naming the copies `original-basilisk.dat` and `original-unicorn.dat`. We can't use:

```
$ cp *.dat original-*.dat
```

because that would expand to:

```
$ cp basilisk.dat minotaur.dat unicorn.dat original-*.dat
```

This wouldn't back up our files, instead we get an error:

Error

```
cp: target ‘original-*.dat’ is not a directory
```

This problem arises when `cp` receives more than two inputs. When this happens, it expects the last input to be a directory where it can copy all the files it was passed. Since there is no directory named `original-*.dat` in the `creatures` directory we get an error.

Instead, we can use a loop: ~~~ $ for filename in *.dat > do > cp $filenameoriginal-$filename > done ~~~

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

```
cp basilisk.dat original-basilisk.dat
```

The second time, the command is:

```
cp minotaur.dat original-minotaur.dat
```

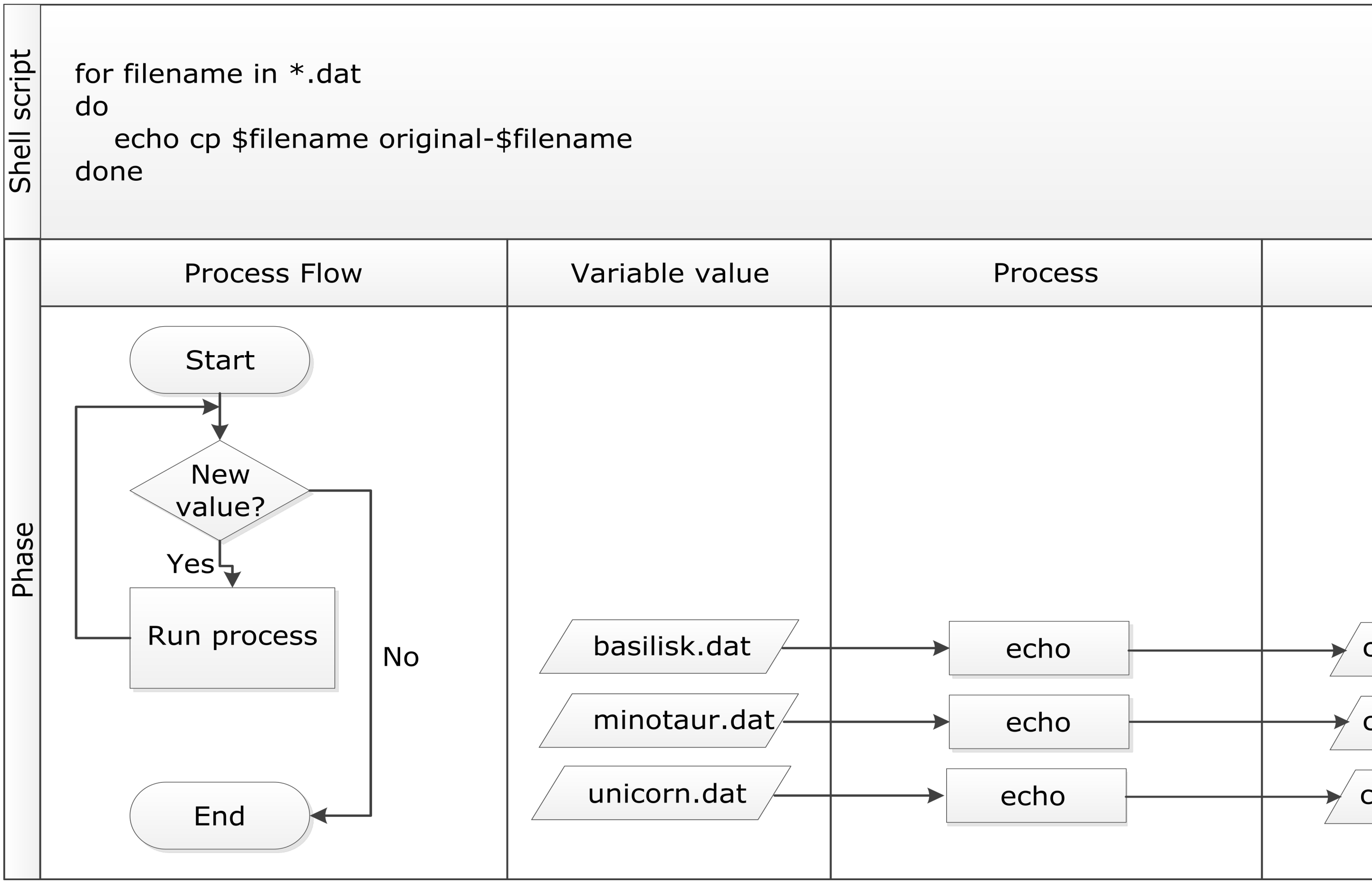
The third and last time, the command is:

```
cp unicorn.dat original-unicorn.dat
```

Since the `cp` command does not normally produce any output, it's hard to check that the loop is doing the correct thing. However, we learned earlier how to print strings using `echo`, and we can modify the loop to use `echo` to print our commands without actually executing them. As such we can check what commands *would be* run in the unmodified loop.

The following diagram shows what happens when the modified loop is executed, and demonstrates how the judicious use of `echo` is a good debugging technique.

The for loop “for filename in *.dat; do echo cp $filenameoriginal-$filename; done″ will successively assign the names of all ″*.dat” files in your current directory to the variable “$filename” and then execute the command. With the files “basilisk.dat”, “minotaur.dat” and “unicorn.dat” in the current directory the loop will successively call the echo command three times and print three lines: “cp basislisk.dat original-basilisk.dat”, then “cp minotaur.dat original-minotaur.dat” and finally “cp unicorn.dat original-unicorn.dat”



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

Nelle's Pipeline: Processing Files

Nelle is now ready to process her data files using `goostats` — a shell script written by her supervisor. This calculates some statistics from a protein sample file, and takes two arguments:

1. an input file (containing the raw data)
2. an output file (to store the calculated statistics)

Since she's still learning how to use the shell, she decides to build up the required commands in stages. Her first step is to make sure that she can select the right input files — remember, these are ones whose names end in 'A' or 'B', rather than 'Z'. Starting from her home directory, Nelle types:

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

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

```
NENE02043A.txt
NENE02043B.txt
```

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

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

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

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

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

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

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

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

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

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

Beginning and End

We can move to the beginning of a line in the shell by typing Ctrl+A and to the end using Ctrl+E.

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

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

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

Those Who Know History Can Choose to Repeat It

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

```
$ history | tail -n 5
```

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

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

Other History Commands

There are a number of other shortcut commands for getting at the history.

- Ctrl+R enters a history search mode 'reverse-i-search' and finds the most recent command in your history that matches the text you enter next. Press Ctrl+R one or more additional times to search for earlier matches. You can then use the left and right arrow keys to choose that line and edit it then hit Return to run the command.
- `!!` retrieves the immediately preceding command (you may or may not find this more convenient than using ↑)
- `!$` retrieves the last word of the last command. That's useful more often than you might expect: after `bash goostats NENE01729B.txt stats-NENE01729B.txt`, you can type `less !$` to look at the file `stats-NENE01729B.txt`, which is quicker than doing ↑ and editing the command-line.

Doing a Dry Run

A loop is a way to do many things at once — or to make many mistakes at once if it does the wrong thing. One way to check what a loop *would* do is to `echo` the commands it would run instead of actually running them.

Suppose we want to preview the commands the following loop will execute without actually running those commands:

```
$ for datafile in *.pdb
> do
>     cat $datafile >> all.pdb
> done
```

What is the difference between the two loops below, and which one would we want to run?

```
# Version 1
$ for datafile in *.pdb
> do
>     echo cat $datafile >> all.pdb
> done
```

```
# Version 2
$ for datafile in *.pdb
> do
>     echo "cat $datafile >> all.pdb"
> done
```

Solution

Solution

The second version is the one we want to run. This prints to screen everything enclosed in the quote marks, expanding the loop variable name because we have prefixed it with a dollar sign.

The first version appends the output from the command `echo cat $datafile` to the file, `all.pdb`. This file will just contain the list; `cat cubane.pdb`, `cat ethane.pdb`, `cat methane.pdb` etc.

Try both versions for yourself to see the output! Be sure to open the `all.pdb` file to view its contents.

Solution

Nested Loops

Suppose we want to set up a directory structure to organize some experiments measuring reaction rate constants with different compounds *and* different temperatures. What would be the result of the following code:

```
$ for species in cubane ethane methane
> do
>     for temperature in 25 30 37 40
>     do
>         mkdir $species-$temperature
>     done
> done
```

Solution

Solution

We have a nested loop, i.e. contained within another loop, so for each species in the outer loop, the inner loop (the nested loop) iterates over the list of temperatures, and creates a new directory for each combination.

Try running the code for yourself to see which directories are created!

Solution

Shell Scripts

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

Let's start by going back to `molecules/` and creating a new file, `middle.sh` which will become our shell script:

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

The command `nano middle.sh` opens the file `middle.sh` within the text editor 'nano' (which runs within the shell). If the file does not exist, it will be created. We can use the text editor to directly edit the file – we'll simply insert the following line:

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

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

Then we save the file (`Ctrl-O` in nano), and exit the text editor (`Ctrl-X` in nano). Check that the directory `molecules` now contains a file called `middle.sh`.

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

```
$ bash middle.sh
```

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

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

Text vs. Whatever

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

What if we want to select lines from an arbitrary file? We could edit `middle.sh` each time to change the filename, but that would probably take longer than typing the command out again in the shell and executing it with a new file name. Instead, let's edit `middle.sh` and make it more versatile:

```
$ nano middle.sh
```

Now, within "nano", replace the text `octane.pdb` with the special variable called `$1`:

```
head -n 15 "$1" | tail -n 5
```

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

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

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

or on a different file like this:

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

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

Double-Quotes Around Arguments

For the same reason that we put the loop variable inside double-quotes, in case the filename happens to contain any spaces, we surround $1 with double-quotes.

Currently, we need to edit middle.sh each time we want to adjust the range of lines that is returned. Let's fix that by configuring our script to instead use three command-line arguments. After the first command-line argument ($1), each additional argument that we provide will be accessible via the special variables $1, $2, $3, which refer to the first, second, third command-line arguments, respectively.

Knowing this, we can use additional arguments to define the range of lines to be passed to head and tail respectively:

```
$ nano middle.sh
```

```
head -n "$2" "$1" | tail -n "$3"
```

We can now run:

```
$ bash middle.sh pentane.pdb 15 5
```

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

By changing the arguments to our command we can change our script's behaviour:

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

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

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

```
$ nano middle.sh
```

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

A comment starts with a `#` character and runs to the end of the line. The computer ignores comments, but they're invaluable for helping people (including your future self) understand and use scripts. The only caveat is that each time you modify the script, you should check that the comment is still accurate: an explanation that sends the reader in the wrong direction is worse than none at all.

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

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

because `wc -l` lists the number of lines in the files (recall that `wc` stands for 'word count', adding the `-l` option means 'count lines' instead) and `sort -n` sorts things numerically. We could put this in a file, but then it would only ever sort a list of `.pdb` files in the current directory. If we want to be able to get a sorted list of other kinds of files, we need a way to get all those names into the script. We can't use `$1`, `$2`, and so on because we don't know how many files there are. Instead, we use the special variable `$@`, which means, 'All of the command-line arguments to the shell script'. We also should put `$@` inside double-quotes to handle the case of arguments containing spaces (`"$@"` is special syntax and is equivalent to `"$1" "$2"` ...).

Here's an example:

```
$ nano sorted.sh
```

```
# Sort files by their length.
# Usage: bash sorted.sh one_or_more_filenames
wc -l "$@" | sort -n
```

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

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

List Unique Species

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

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

An example of this type of file is given in `data-shell/data/animal-counts/animals.txt`.

We can use the command `cut -d , -f 2 animals.txt | sort | uniq` to produce the unique species in `animals.txt`. In order to avoid having to type out this series of commands every time, a scientist may choose to write a shell script instead.

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

Solution

Solution

```
# Script to find unique species in csv files where species is the second data field
# This script accepts any number of file names as command line arguments

# Loop over all files
for file in $@
do
    echo "Unique species in $file:"
    # Extract species names
    cut -d , -f 2 $file | sort | uniq
done
```

Solution

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

```
$ history | tail -n 5 > redo-figure-3.sh
```

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

```
297 bash goostats NENE01729B.txt stats-NENE01729B.txt
298 bash goodiff stats-NENE01729B.txt /data/validated/01729.txt > 01729-differences.txt
299 cut -d ',' -f 2-3 01729-differences.txt > 01729-time-series.txt
300 ygraph --format scatter --color bw --borders none 01729-time-series.txt figure-3.png
301 history | tail -n 5 > redo-figure-3.sh
```

After a moment's work in an editor to remove the serial numbers on the commands, and to remove the final line where we called the `history` command, we have a completely accurate record of how we created that figure.

Why Record Commands in the History Before Running Them?

If you run the command:

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

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

Solution

Solution

If a command causes something to crash or hang, it might be useful to know what that command was, in order to investigate the problem. Were the command only be recorded after running it, we would not have a record of the last command run in the event of a crash.

Solution

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

Nelle's Pipeline: Creating a Script

Nelle's supervisor insisted that all her analytics must be reproducible. The easiest way to capture all the steps is in a script.

First we return to Nelle's data directory:

```
$ cd ../north-pacific-gyre/2012-07-03/
```

She runs the editor and writes the following:

```
# Calculate stats for data files.
for datafile in "$@"
do
    echo $datafile
    bash goostats $datafile stats-$datafile
done
```

She saves this in a file called `do-stats.sh` so that she can now re-do the first stage of her analysis by typing:

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

She can also do this:

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

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

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

```
# Calculate stats for Site A and Site B data files.
for datafile in NENE*[AB].txt
do
    echo $datafile
    bash goostats $datafile stats-$datafile
done
```

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

Debugging Scripts

Suppose you have saved the following script in a file called `do-errors.sh` in Nelle's `north-pacific-gyre/2012-07-03` directory:

```
# Calculate stats for data files.
for datafile in "$@"
do
    echo $datfile
    bash goostats $datafile stats-$datafile
done
```

When you run it:

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

the output is blank. To figure out why, re-run the script using the `-x` option:

```
bash -x do-errors.sh NENE*[AB].txt
```

What is the output showing you? Which line is responsible for the error?

Solution

Solution

The `-x` option causes `bash` to run in debug mode. This prints out each command as it is run, which will help you to locate errors. In this example, we can see that `echo` isn't printing anything. We have made a typo in the loop variable name, and the variable `datfile` doesn't exist, hence returning an empty string.

Solution

Finding Things

In the same way that many of us now use ‘Google’ as a verb meaning ‘to find’, Unix programmers often use the word ‘grep’. ‘grep’ is a contraction of ‘global/regular expression/print’, a common sequence of operations in early Unix text editors. It is also the name of a very useful command-line program.

`grep` finds and prints lines in files that match a pattern. For our examples, we will use a file that contains three haikus taken from a 1998 competition in *Salon* magazine. For this set of examples, we’re going to be working in the writing subdirectory:

```
$ cd
$ cd Desktop/data-shell/writing
$ cat haiku.txt
```

```
The Tao that is seen
Is not the true Tao, until
You bring fresh toner.

With searching comes loss
and the presence of absence:
"My Thesis" not found.

Yesterday it worked
Today it is not working
Software is like that.
```

> ### Forever, or Five Years
>
> We haven’t linked to the original haikus because they don’t appear to be on *Salon*’s site any longer. As Jeff Rothenberg said, ‘Digital information lasts forever — or five years, whichever comes first.’ Luckily, popular content often has backups.

Let’s find lines that contain the word ‘not’:

```
$ grep not haiku.txt
```

```
Is not the true Tao, until
"My Thesis" not found
Today it is not working
```

Here, `not` is the pattern we’re searching for. The grep command searches through the file, looking for matches to the pattern specified. To use it type `grep`, then the pattern we’re searching for and finally the name of the file (or files) we’re searching in.

The output is the three lines in the file that contain the letters ‘not’.

By default, grep searches for a pattern in a case-sensitive way. In addition, the search pattern we have selected does not have to form a complete word, as we will see in the next example.

Let’s search for the pattern: ‘The’.

```
$ grep The haiku.txt
```

```
The Tao that is seen
"My Thesis" not found.
```

This time, two lines that include the letters 'The' are outputted, one of which contained our search pattern within a larger word, 'Thesis'.

To restrict matches to lines containing the word 'The' on its own, we can give `grep` with the `-w` option. This will limit matches to word boundaries.

Later in this lesson, we will also see how we can change the search behavior of grep with respect to its case sensitivity.

```
$ grep -w The haiku.txt
```

```
The Tao that is seen
```

Note that a 'word boundary' includes the start and end of a line, so not just letters surrounded by spaces. Sometimes we don't want to search for a single word, but a phrase. This is also easy to do with `grep` by putting the phrase in quotes.

```
$ grep -w "is not" haiku.txt
```

```
Today it is not working
```

We've now seen that you don't have to have quotes around single words, but it is useful to use quotes when searching for multiple words. It also helps to make it easier to distinguish between the search term or phrase and the file being searched. We will use quotes in the remaining examples.

Another useful option is `-n`, which numbers the lines that match:

```
$ grep -n "it" haiku.txt
```

```
5:With searching comes loss
9:Yesterday it worked
10:Today it is not working
```

Here, we can see that lines 5, 9, and 10 contain the letters 'it'.

We can combine options (i.e. flags) as we do with other Unix commands. For example, let's find the lines that contain the word 'the'. We can combine the option `-w` to find the lines that contain the word 'the' and `-n` to number the lines that match:

```
$ grep -n -w "the" haiku.txt
```

```
2:Is not the true Tao, until
6:and the presence of absence:
```

Now we want to use the option `-i` to make our search case-insensitive:

```
$ grep -n -w -i "the" haiku.txt
```

```
1:The Tao that is seen
2:Is not the true Tao, until
6:and the presence of absence:
```

Now, we want to use the option `-v` to invert our search, i.e., we want to output the lines that do not contain the word 'the'.

```
$ grep -n -w -v "the" haiku.txt
```

```
1:The Tao that is seen
3:You bring fresh toner.
4:
5:With searching comes loss
7:"My Thesis" not found.
8:
9:Yesterday it worked
10:Today it is not working
11:Software is like that.
```

If we use the `-r` (recursive) option, `grep` can search for a pattern recursively through a set of files in subdirectories.

Let's search recursively for `Yesterday` in the `data-shell/writing` directory:

```
$ grep -r Yesterday .
```

```
data/LittleWomen.txt:"Yesterday, when Aunt was asleep and I was trying to be as still as a
data/LittleWomen.txt:Yesterday at dinner, when an Austrian officer stared at us and then
data/LittleWomen.txt:Yesterday was a quiet day spent in teaching, sewing, and writing in my
haiku.txt:Yesterday it worked
```

`grep` has lots of other options. To find out what they are, we can type:

```
$ grep --help
```

```
Usage: grep [OPTION]... PATTERN [FILE]...
Search for PATTERN in each FILE or standard input.
PATTERN is, by default, a basic regular expression (BRE).
Example: grep -i 'hello world' menu.h main.c

Regexp selection and interpretation:
  -E, --extended-regexp     PATTERN is an extended regular expression (ERE)
  -F, --fixed-strings       PATTERN is a set of newline-separated fixed strings
  -G, --basic-regexp        PATTERN is a basic regular expression (BRE)
  -P, --perl-regexp         PATTERN is a Perl regular expression
  -e, --regexp=PATTERN      use PATTERN for matching
  -f, --file=FILE           obtain PATTERN from FILE
  -i, --ignore-case         ignore case distinctions
  -w, --word-regexp         force PATTERN to match only whole words
  -x, --line-regexp         force PATTERN to match only whole lines
  -z, --null-data           a data line ends in 0 byte, not newline

Miscellaneous:
...        ...        ...
```

Using grep

Which command would result in the following output:

```
and the presence of absence:
```

1. `grep "of" haiku.txt`
2. `grep -E "of" haiku.txt`
3. `grep -w "of" haiku.txt`
4. `grep -i "of" haiku.txt`

Solution

Solution

The correct answer is 3, because the `-w` option looks only for whole-word matches. The other options will also match 'of' when part of another word.

Solution

Wildcards

`grep`'s real power doesn't come from its options, though; it comes from the fact that patterns can include wildcards. (The technical name for these is **regular expressions**, which is what the 're' in 'grep' stands for.) Regular expressions are both complex and powerful; if you want to do complex searches, please look at the lesson on our website. As a taster, we can find lines that have an 'o' in the second position like this:

```
$ grep -E "^.o" haiku.txt
```

```
You bring fresh toner.
Today it is not working
Software is like that.
```

We use the `-E` option and put the pattern in quotes to prevent the shell from trying to interpret it. (If the pattern contained a `*`, for example, the shell would try to expand it before running `grep`.) The `^` in the pattern anchors the match to the start of the line. The `.` matches a single character (just like `?` in the shell), while the `o` matches an actual 'o'.

Tracking a Species

Leah has several hundred data files saved in one directory, each of which is formatted like this:

```
2013-11-05,deer,5
2013-11-05,rabbit,22
2013-11-05,raccoon,7
2013-11-06,rabbit,19
2013-11-06,deer,2
```

She wants to write a shell script that takes a species as the first command-line argument and a directory as the second argument. The script should return one file called `species.txt` containing a list of dates and the number of that species seen on each date. For example using the data shown above, `rabbit.txt` would contain:

```
2013-11-05,22
2013-11-06,19
```

Put these commands and pipes in the right order to achieve this:

```
cut -d : -f 2
>
|
grep -w $1 -r $2
|
$1.txt
cut -d , -f 1,3
```

Hint: use `man grep` to look for how to grep text recursively in a directory and `man cut` to select more than one field in a line.

An example of such a file is provided in `data-shell/data/animal-counts/animals.txt`

Solution

Solution

```
grep -w $1 -r $2 | cut -d : -f 2 | cut -d , -f 1,3  > $1.txt
```

You would call the script above like this:

```
$ bash count-species.sh bear .
```

Solution

Little Women

You and your friend, having just finished reading *Little Women* by Louisa May Alcott, are in an argument. Of the four sisters in the book, Jo, Meg, Beth, and Amy, your friend thinks that Jo was the most mentioned. You, however, are certain it was Amy. Luckily, you have a file `LittleWomen.txt` containing the full text of the novel (`data-shell/writing/data/LittleWomen.txt`). Using a `for` loop, how would you tabulate the number of times each of the four sisters is mentioned?

Hint: one solution might employ the commands `grep` and `wc` and a `|`, while another might utilize `grep` options. There is often more than one way to solve a programming task, so a particular solution is usually chosen based on a combination of yielding the correct result, elegance, readability, and speed.

Solution

Solutions

```
for sis in Jo Meg Beth Amy
do
    echo $sis:
grep -ow $sis LittleWomen.txt | wc -l
done
```

Alternative, slightly inferior solution:

```
for sis in Jo Meg Beth Amy
do
    echo $sis:
grep -ocw $sis LittleWomen.txt
done
```

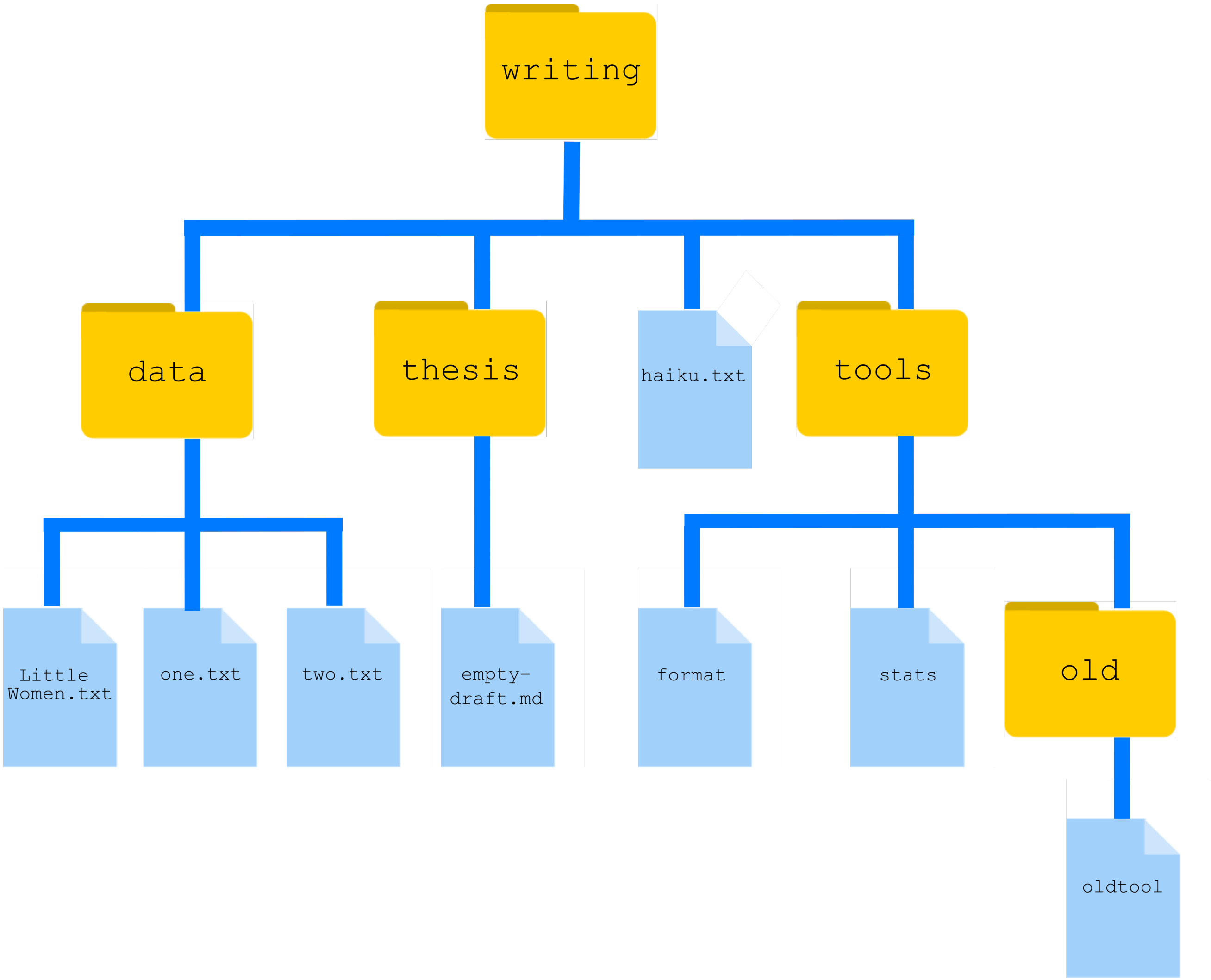
This solution is inferior because `grep -c` only reports the number of lines matched. The total number of matches reported by this method will be lower if there is more than one match per line.

Perceptive observers may have noticed that character names sometimes appear in all-uppercase in chapter titles (e.g. 'MEG GOES TO VANITY FAIR'). If you wanted to count these as well, you could add the `-i` option for case-insensitivity (though in this case, it doesn't affect the answer to which sister is mentioned most frequently).

Solution

While `grep` finds lines in files, the `find` command finds files themselves. Again, it has a lot of options; to show how the simplest ones work, we'll use the directory tree shown below.

A file tree under the directory "writing" contians several sub-directories and files such that "writing" contains directories "data", "thesis", "tools" and a file "haiku.txt"; "writing/data" contains the files "Little Women.txt", "one.txt" and "two.txt"; "writing/thesis" contains the file "empty-draft.md"; "writing/tools" contains the directory "old" and the files "format" and "stats"; and "writing/tools/old" contains a file "old-tool"



Nelle's `writing` directory contains one file called `haiku.txt` and three subdirectories: `thesis` (which contains a sadly empty file, `empty-draft.md`); `data` (which contains three files `LittleWomen.txt`, `one.txt` and `two.txt`); and a `tools` directory that contains the programs `format` and `stats`, and a subdirectory called `old`, with a file `oldtool`.

For our first command, let's run `find .` (remember to run this command from the `data-shell/writing` folder).

```
$ find .
```

```
.
./data
./data/one.txt
./data/LittleWomen.txt
./data/two.txt
./tools
./tools/format
./tools/old
./tools/old/oldtool
./tools/stats
./haiku.txt
```

```
./thesis
./thesis/empty-draft.md
```

As always, the `.` on its own means the current working directory, which is where we want our search to start. `find`'s output is the names of every file **and** directory under the current working directory. This can seem useless at first but `find` has many options to filter the output and in this lesson we will discover some of them.

The first option in our list is `-type d` that means 'things that are directories'. Sure enough, `find`'s output is the names of the five directories in our little tree (including `.`):

```
$ find . -type d
```

```
./
./data
./thesis
./tools
./tools/old
```

Notice that the objects `find` finds are not listed in any particular order. If we change `-type d` to `-type f`, we get a listing of all the files instead:

```
$ find . -type f
```

```
./haiku.txt
./tools/stats
./tools/old/oldtool
./tools/format
./thesis/empty-draft.md
./data/one.txt
./data/LittleWomen.txt
./data/two.txt
```

Now let's try matching by name:

```
$ find . -name *.txt
```

```
./haiku.txt
```

We expected it to find all the text files, but it only prints out `./haiku.txt`. The problem is that the shell expands wildcard characters like `*` *before* commands run. Since `*.txt` in the current directory expands to `haiku.txt`, the command we actually ran was:

```
$ find . -name haiku.txt
```

`find` did what we asked; we just asked for the wrong thing.

To get what we want, let's do what we did with `grep`: put `*.txt` in quotes to prevent the shell from expanding the `*` wildcard. This way, `find` actually gets the pattern `*.txt`, not the expanded filename `haiku.txt`:

```
$ find . -name "*.txt"
```

```
./data/one.txt
./data/LittleWomen.txt
./data/two.txt
./haiku.txt
```

Listing vs. Finding

`ls` and `find` can be made to do similar things given the right options, but under normal circumstances, `ls` lists everything it can, while `find` searches for things with certain properties and shows them.

As we said earlier, the command line's power lies in combining tools. We've seen how to do that with pipes; let's look at another technique. As we just saw, `find . -name "*.txt"` gives us a list of all text files in or below the current directory. How can we combine that with `wc -l` to count the lines in all those files?

The simplest way is to put the `find` command inside `$()`:

```
$ wc -l $(find . -name "*.txt")
```

```
11 ./haiku.txt
300 ./data/two.txt
21022 ./data/LittleWomen.txt
70 ./data/one.txt
21403 total
```

When the shell executes this command, the first thing it does is run whatever is inside the `$()`. It then replaces the `$()` expression with that command's output. Since the output of `find` is the four filenames `./data/one.txt`, `./data/LittleWomen.txt`, `./data/two.txt`, and `./haiku.txt`, the shell constructs the command:

```
$ wc -l ./data/one.txt ./data/LittleWomen.txt ./data/two.txt ./haiku.txt
```

which is what we wanted. This expansion is exactly what the shell does when it expands wildcards like `*` and `?`, but lets us use any command we want as our own 'wildcard'.

It's very common to use `find` and `grep` together. The first finds files that match a pattern; the second looks for lines inside those files that match another pattern. Here, for example, we can find PDB files that contain iron atoms by looking for the string 'FE' in all the `.pdb` files above the current directory:

```
$ grep "FE" $(find .. -name "*.pdb")
```

```
../data/pdb/heme.pdb:ATOM     25 FE           1      -0.924   0.535  -0.518
```

Matching and Subtracting

The `-v` option to `grep` inverts pattern matching, so that only lines which do *not* match the pattern are printed. Given that, which of the following commands will find all files in `/data` whose names end in `s.txt` but whose names also do *not* contain the string `net`? (For example, `animals.txt` or `amino-acids.txt` but not `planets.txt`.) Once you have thought about your answer, you can test the commands in the `data-shell` directory.

1. `find data -name "*s.txt" | grep -v net`
2. `find data -name *s.txt | grep -v net`
3. `grep -v "net" $(find data -name "*s.txt")`
4. None of the above.

Solution

Solution

The correct answer is 1. Putting the match expression in quotes prevents the shell expanding it, so it gets passed to the `find` command.

Option 2 is incorrect because the shell expands `*s.txt` instead of passing the wildcard expression to `find`.

Option 3 is incorrect because it searches the contents of the files for lines which do not match 'net', rather than searching the file names.

Solution

Binary Files

We have focused exclusively on finding patterns in text files. What if your data is stored as images, in databases, or in some other format?

A handful of tools extend `grep` to handle a few non text formats. But a more generalizable approach is to convert the data to text, or extract the text-like elements from the data. On the one hand, it makes simple things easy to do. On the other hand, complex things are usually impossible. For example, it's easy enough to write a program that will extract X and Y dimensions from image files for `grep` to play with, but how would you write something to find values in a spreadsheet whose cells contained formulas?

A last option is to recognize that the shell and text processing have their limits, and to use another programming language. When the time comes to do this, don't be too hard on the shell: many modern programming languages have borrowed a lot of ideas from it, and imitation is also the sincerest form of praise.

The Unix shell is older than most of the people who use it. It has survived so long because it is one of the most productive programming environments ever created — maybe even *the* most productive. Its syntax may be cryptic, but people who have mastered it can experiment with different commands interactively, then use what they have learned to automate their work. Graphical user interfaces may be better at the first, but the shell is still unbeaten at the second. And as Alfred North Whitehead wrote in 1911, 'Civilization advances by extending the number of important operations which we can perform without thinking about them.'

`find` Pipeline Reading Comprehension

Write a short explanatory comment for the following shell script:

```
wc -l $(find . -name "*.dat") | sort -n
```

Solution

Solution

1. Find all files with a `.dat` extension recursively from the current directory
2. Count the number of lines each of these files contains
3. Sort the output from step 2. numerically

Solution

Transferring Files

There are other ways to interact with remote files other than git.

It is true that we can clone an entire git repository, or even one level of a git repository using: `git clone --depth-1 repository_name`. What about files that do not exist in a git repository? If we wish to download files from the shell we can use tools such as Wget and cURL.

Wget

Wget is a simple tool developed for the GNU Project that downloads files with the HTTP, HTTPS and FTP protocols. It is widely used by Unix-like users and is available with most Linux distributions.

To download the fastq file used in week one (located at https://raw.githubusercontent.com/BayLab/MarineGenomicsData/main/Week1/SRR6805880_1.fastq) from the web via HTTP we can simply type:

Bash

```
$ wget https://raw.githubusercontent.com/BayLab/MarineGenomicsData/main/Week1/SRR6805880_1.fastq
```

Output

```
--2021-04-01 08:06:38--  https://raw.githubusercontent.com/BayLab/MarineGenomicsData/main/Week1/SRR68058
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.111.133, 185.199.110.133, 18
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.111.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 250890 (245K) [text/plain]
Saving to: ‘SRR6805880_1.fastq’

SRR6805880_1.fastq               100%[=======================================================>] 245.01K

2021-04-01 08:06:38 (4.61 MB/s) - ‘SRR6805880_1.fastq’ saved [250890/250890]


## cURL

Alternatively, we can use ‘cURL‘.
It supports a much larger range of protocols including common mail based protocols like pop3 and smtp.

Let’s use cURL to download the second fastq file (located at https://raw.githubusercontent.com/BayLab/M
from the web via HTTP we can simply type:

Bash
```

$ curl -o SRR6805880_2.fastq https://raw.githubusercontent.com/BayLab/MarineGenomicsData/main/Week1/SRR6805880_2.fastq ~~~

Output

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  245k  100  245k    0     0  2087k      0 --:--:-- --:--:-- --:--:-- 2076k
```

This input to curl is in the form:

Bash

```
curl -o filename_for_local_machine target_url
```

where the `-o` option says write the output to a file instead of the stdout (the screen), and file_name_for_local_machine is any file name you choose to save to the local machine, and target_URL is where the file is the URL where the file is on the web

Removing the `-o` option, and following the syntax `curl target_URL` outputs the contents of the url to the screen.

`Wget` offers more functionality natively than `curl` for retrieving entire directories. We could use `Wget` to first retrieve an entire directory and then run `html2text` and `grep` to find a particular string. `cURL` is limited to retrieving one or more specified URLs that cannot be obtained by recursively crawling a directory. The situation may be improved by combining with other unix tools, but is not thought as being as good as `Wget`.

Please refer to the man pages by typing `man wget`, `man curl`, and `man html2text` in the shell for more information.

Permission

Unix controls who can read, modify, and run files using *permissions*. We'll discuss how Windows handles permissions at the end of the section: the concepts are similar, but the rules are different.

Let's look at files and directories. Every file and directory on a Unix computer belongs to one owner and one group. Along with each file's content, the operating system stores the numeric IDs of the user and group that own it.

The user-and-group model means that for each file every user on the system falls into one of three categories: the owner of the file, someone in the file's group, and everyone else.

For each of these three categories, the computer keeps track of whether people in that category can read the file, write to the file, or execute the file (i.e., run it if it is a program).

For example, if a file had the following set of permissions:

user

group

all

read

yes

yes

no

write

yes

no

no

execute

no

no

no

it would mean that:

- the file’s owner can read and write it, but not run it;
- other people in the file’s group can read it, but not modify it or run it; and
- everybody else can do nothing with it at all.

Let’s run the command `ls -l` in /data-shell/molecules :

Bash

```
$ ls -l
```

Output

```
-rwxr-x--- 1 vguerra vguerra 1.2K Apr  1 05:43 cubane.pdb
-rwxr-x--- 1 vguerra vguerra  622 Apr  1 05:43 ethane.pdb
-rwxr-x--- 1 vguerra vguerra  422 Apr  1 05:43 methane.pdb
-rwxr-x--- 1 vguerra vguerra 1.8K Apr  1 05:43 octane.pdb
-rwxr-x--- 1 vguerra vguerra 1.2K Apr  1 05:43 pentane.pdb
-rwxr-x--- 1 vguerra vguerra  825 Apr  1 05:43 propane.pdb
```

The `-l` flag tells `ls` to give us a long-form listing. It’s a lot of information, so let’s go through the columns in turn.

On the right side, we have the files’ names. Next to them, moving left, are the times and dates they were last modified. Backup systems and other tools use this information in a variety of ways, but you can use it to tell when you (or anyone else with permission) last changed a file.

Next to the modification time is the file’s size in bytes and the names of the user and group that owns it (in this case, `vlad` and `bio` respectively). We’ll skip over the second column for now (the one showing `1` for each file) because it’s the first column that we care about most. This shows the file’s permissions, i.e., who can read, write, or execute it.

Let’s have a closer look at one of those permission strings: `-rwxr-x---`. The first character tells us what type of thing this is: ‘-’ means it’s a regular file, while ‘d’ means it’s a directory, and other characters mean more esoteric things.

The next three characters tell us what permissions the file’s owner has. Here, the owner can read, write, and execute the file: `rwx`. The middle triplet shows us the group’s permissions. If the permission is turned off, we see a dash, so `r-x` means “read and execute, but not write”. The final triplet shows us what everyone who isn’t the file’s owner, or in the file’s group, can do. In this case, it’s ‘–’ , so nobody on the system can look at the file’s contents and run it.

To change permissions, we use the `chmod` command (whose name stands for “change mode”).

You can share your files with other users in your system that are not in your group by changing the final triplet. Let’s change the permission of the file propane.pdb

Bash

```
$ ls -l propane.pdb
```

Output

```
-rwxr-x--- 1 vguerra vguerra  825 Apr  1 05:43 propane.pdb
```

We can make the change with `chmod` . The program can change the permission of the user (u), group (g), and/or other (o). Let's add reading (r), writing (x), and execute (x) permissions to the other group.

Bash

```
$ chmod o=rwx propane.pdb
```

Output

```
-rwxr-xrwx 1 vguerra vguerra  825 Apr  1 05:43 propane.pdb
```

We will not cover this today, but the ownership of the file could also be changed with 'chgrp'. This command is useful when you have to store files in shared disks or when you have to manage the space you are allowed to use in a cluster.

What about Windows?

Those are the basics of permissions on Unix. As we said at the outset, though, things work differently on Windows. There, permissions are defined by access control lists, or ACLs. An ACL is a list of pairs, each of which combines a "who" with a "what". For example, you could give the Mummy permission to append data to a file without giving him permission to read or delete it, and give Frankenstein permission to delete a file without being able to see what it contains.

This is more flexible that the Unix model, but it's also more complex to administer and understand on small systems. (If you have a large computer system, *nothing* is easy to administer or understand.) Some modern variants of Unix support ACLs as well as the older read-write-execute permissions, but hardly anyone uses them.

Awk

If we need to count the number of lines in a file, we can use the previously showed command for word counting wc

Bash

```
$ wc -l propane.pdb
```

Output

```
15 propane.pdb
```

As you probably remember, -l is an option that asks for the number of lines only.

However, wc counts the number of newlines in the file, if the last line does not contain a carriage return (i.e. there is no emptyline at the end of the file), the result is going be the actual number of lines minus one.

A workaround is to use Awk. Awk is command line program that takes as input a set of instructions and one or more files. The instructions are executed on each line of the input file(s).

The instructions are enclosed in single quotes or they can be read from a file.

Example:

Bash

```
$ awk '{print $0}' propane.pdb
```

Output

```
COMPND      PROPANE
AUTHOR      DAVE WOODCOCK  95  12 18
ATOM      1  C           1       1.241   0.444   0.349  1.00  0.00
ATOM      2  C           1      -0.011  -0.441   0.333  1.00  0.00
ATOM      3  C           1      -1.176   0.296  -0.332  1.00  0.00
ATOM      4  H           1       1.516   0.699  -0.675  1.00  0.00
ATOM      5  H           1       2.058  -0.099   0.827  1.00  0.00
ATOM      6  H           1       1.035   1.354   0.913  1.00  0.00
ATOM      7  H           1      -0.283  -0.691   1.359  1.00  0.00
ATOM      8  H           1       0.204  -1.354  -0.225  1.00  0.00
ATOM      9  H           1      -0.914   0.551  -1.359  1.00  0.00
ATOM     10  H           1      -1.396   1.211   0.219  1.00  0.00
ATOM     11  H           1      -2.058  -0.345  -0.332  1.00  0.00
TER      12              1
END
```

This command has the same output of “cat”: it prints each line from the example.fasta file.

The structure of the instruction is the following: - curly braces surround the set of instructions - print is the instruction that sends its arguments to the terminal - $0 is a variable, it means “the content of the current line”

As you can see, the file contains a table.

Awk automatically splits the processed line by looking at spaces: in our case it has knowledge of the different columns in the table.

Each column value for the current line is stored into a variable: $1 for the first column, $2 for the second and so on.

So, if we like to print only the second column from the table, we execute

Bash

```
$ awk '{print $2}' propane.pdb
```

Output

```
PROPANE
DAVE
1
2
3
4
5
6
7
8
9
10
11
12
```

We can also print more than one value, or add text to the printed line:

Bash

```
$ awk '{print "chr",$2,$3}' propane.pdb
```

Output

```
chr PROPANE
chr DAVE WOODCOCK
chr 1 C
chr 2 C
chr 3 C
chr 4 H
chr 5 H
chr 6 H
chr 7 H
chr 8 H
chr 9 H
chr 10 H
chr 11 H
chr 12 1
chr
```

The comma puts a space between the printed values. Strings of text should be enclosed in double quotes. In this case we are printing the text “chr”, the second and the fourth column for each row in the table.

So, $0 is the whole line, $1 the first field, $2 the second and so on. What if we want to print the last column, but we don’t know its number? Maybe it is a huge table, or maybe different lines have a different number of columns.

Awk helps us thanks to the variable NF. NF stores the number of fields (our columns) in the row. Let’s see for our table:

Bash

```
$ awk '{print NF}' propane.pdb
```

Outline

```
2
6
9
9
9
9
9
9
9
9
9
9
9
3
1
```

We can see how many fields are store in each line. The second line contains 6, and the third contains 9. Since NF is the number of the last field, $NF contains its value.

Bash

```
$ awk '{print "This line has",NF,"columns. The last one contains",$NF}' propane.pdb
```

Output

```
This line has 2 columns. The last one contains PROPANE
This line has 6 columns. The last one contains 18
This line has 9 columns. The last one contains 0.00
This line has 9 columns. The last one contains 0.00
This line has 9 columns. The last one contains 0.00
This line has 9 columns. The last one contains 0.00
This line has 9 columns. The last one contains 0.00
This line has 9 columns. The last one contains 0.00
This line has 9 columns. The last one contains 0.00
This line has 9 columns. The last one contains 0.00
This line has 9 columns. The last one contains 0.00
This line has 9 columns. The last one contains 0.00
This line has 9 columns. The last one contains 0.00
This line has 3 columns. The last one contains 1
This line has 1 columns. The last one contains END
```

Compare this output to the content of the file propane.pdb

Bash

```
$ cat propane.pdb
```

Output

```
COMPND      PROPANE
AUTHOR      DAVE WOODCOCK  95  12 18
ATOM      1  C           1       1.241   0.444   0.349  1.00  0.00
ATOM      2  C           1      -0.011  -0.441   0.333  1.00  0.00
ATOM      3  C           1      -1.176   0.296  -0.332  1.00  0.00
ATOM      4  H           1       1.516   0.699  -0.675  1.00  0.00
ATOM      5  H           1       2.058  -0.099   0.827  1.00  0.00
ATOM      6  H           1       1.035   1.354   0.913  1.00  0.00
ATOM      7  H           1      -0.283  -0.691   1.359  1.00  0.00
ATOM      8  H           1       0.204  -1.354  -0.225  1.00  0.00
ATOM      9  H           1      -0.914   0.551  -1.359  1.00  0.00
ATOM     10  H           1      -1.396   1.211   0.219  1.00  0.00
ATOM     11  H           1      -2.058  -0.345  -0.332  1.00  0.00
TER      12              1
END
```

Field separator

Out there we have different file formats: our data may be comma separated (csv), tab separated (tsv), by semicolon or by any other character.

Create the a csv file called choice.all.csv with nano(from Bay et al. 2017 https://www.cell.com/current-biology/pdf/S0960-9822(17)31202-2.pdf) :

choice.all.csv

```
id,pedigree.ID,sourcepond,avg.choice
09_A1_F2F_001,503_F2F_001_G03,pond7,0
09_A1_F2F_004,506_F2F_004_B04,pond7,1
09_A1_F2F_005,507_F2F_005_C04,pond7,1
09_A1_F2F_009,511_F2F_009_G04,pond7,0
```

To specify the field separator, we should provide it a command line like:

Bash

```
$ awk -F "," ’{print $2}’ choice.all.csv
```

In this case , we are printing the second field in each line, using comma as separator. Please notice that the character space is now part of the field value, since it is no longer the separator.

Matching lines

Maybe we would like to perform different instruction on different lines.

Awk allows you to specify a matching pattern, like the command grep does.

Let’s look at the file content

Bash

```
$ awk ’{print $0}’ propane.pdb
```

It seems an abriged PDB file. If we would like to print only lines starting with the word “ATOM”, we type:

Bash

```
$ awk ’/^ATOM/ {print $0}’ propane.pdb
```

Output

```
ATOM      1  C           1       1.241   0.444   0.349  1.00  0.00
ATOM      2  C           1      -0.011  -0.441   0.333  1.00  0.00
ATOM      3  C           1      -1.176   0.296  -0.332  1.00  0.00
ATOM      4  H           1       1.516   0.699  -0.675  1.00  0.00
ATOM      5  H           1       2.058  -0.099   0.827  1.00  0.00
ATOM      6  H           1       1.035   1.354   0.913  1.00  0.00
ATOM      7  H           1      -0.283  -0.691   1.359  1.00  0.00
ATOM      8  H           1       0.204  -1.354  -0.225  1.00  0.00
ATOM      9  H           1      -0.914   0.551  -1.359  1.00  0.00
ATOM     10  H           1      -1.396   1.211   0.219  1.00  0.00
ATOM     11  H           1      -2.058  -0.345  -0.332  1.00  0.00
```

In this case, we specify the pattern before the instructions: only lines starting with the text "ATOM". As you remember, ^ means "at the beginning of the line".

We can print more than one column:

Bash

```
$ awk '/^ATOM/ {print $7,$8,$9}' propane.pdb
```

In this case, we are printing the spatial coordinates of each atom.

Awk is a powerful program that deserves more time. It replaces many of the programs that we have talked about today and in the past lesson.

Stream editor

Another useful program for bioinformatics is `sed`. It is a great program to use to change headers of fastq files.

Sed can replace a string of characters or add characters when needed.

Let's add a character at the end of the fastq files SRR6805880 headers. If we wanted to do a differential expressiona analysis in a program like Trinity we would have to change the header of this files so that each header ends in `\1` or `\2` (go back to the folder where we have the fastq files SRR6805880_1.fastq and SRR6805880_2.fastq)

Let's look again at the SRR6805880_1.fastq

```
@SRR6805880.1 9F8KO:01339:11608 length=80
TGCAGGAGGGCCCATCAAAATGGTATTAATCATGAGAAAAGAGACAAGGATAATTCATTTTGCGGTACACCAGTTAGACA
+SRR6805880.1 9F8KO:01339:11608 length=80
04789157709<2;:8888-8:5:;;7<5:<<<<;:>>?1;<;>>=5:59:;5;496665,656-1367959893789:8
@SRR6805880.3 9F8KO:01356:11636 length=80
TGCAGATAAACAAAAGTATAGGATATATGCAACTGTAACAGCGCTGTAAATAAACAACAGTATACAGGCAACTGTAACAG
+SRR6805880.3 9F8KO:01356:11636 length=80
37558:;>=24554+1579996;;;::9999577689088:9998///828892:9497879998872774879793444
```

We are going to replace the last two columns of the headers with `\1` using sed `'s/matching/replacing/' file.fastq`

Bash

```
$ sed '/^@SRR6805880/ s/ .*/\/1/' SRR6805880_1.fastq
```

Output

```
@SRR6805880.1/1
TGCAGGAGGGCCCATCAAAATGGTATTAATCATGAGAAAAGAGACAAGGATAATTCATTTTGCGGTACACCAGTTAGACA
+SRR6805880.1 9F8KO:01339:11608 length=80
04789157709<2;:8888-8:5:;;7<5:<<<<;:>>?1;<;>>=5:59:;5;496665,656-1367959893789:8
@SRR6805880.3/1
TGCAGATAAACAAAAGTATAGGATATATGCAACTGTAACAGCGCTGTAAATAAACAACAGTATACAGGCAACTGTAACAG
+SRR6805880.3 9F8KO:01356:11636 length=80
37558:;>=24554+1579996;;;::9999577689088:9998///828892:9497879998872774879793444
```

If time allows, we will explore more sed options. But if not, there are a lot of online resources to learn more about regular expressions: https://en.wikipedia.org/wiki/Regular_expression