Getting Comfortable with the Command Line

a bioinformatics tutorial

Matt Johnson

June 23, 2015

1 Introduction

1.1 What is the Command Line?

For most of their history, the only way to interact with computers was to type commands into a terminal. Today, most operating systems and applications use Graphical User Interfaces (GUIs) to present information and record input from a user. However, many operations on a computer can still be conducted at the command line.

There are two primary reasons why getting comfortable using the command line is useful in bioinformatics. First, the development of a GUI is difficult and many bioinformatics tools simply lack them. Second, as you become more experienced, you will find it easier to write your own programs for achieving specific tasks, rather than relying on someone else's code.

The goal of this tutorial is to provide you with the basic tools for dealing with the command line in a UNIX-like environment.

In this tutorial, commands you can type are represented by `block text`.

1.2 Starting up the Terminal

The MacOSX operating system is based on UNIX, so it comes pre-installed with a UNIX command line program called Terminal. It is located within the Utilities folder, within the Applications folder on your computer. Drag the Terminal app to your dock, then click to open.

Windows, by contrast, is not a UNIX-based operating system, so you will need to install your own terminal. Cygwin is recommended: `http://www.cygwin.com`

Whichever program you open, you will be greeted by a window that looks something like this:

```
Last login: Thu Sep 5 14:45:49 on ttys004
CBG002827:~ mjohnson$
```

In this MacOSX terminal window, you can see that opening the program started a process `bash`. Bash is a type of *shell*, an interface for typing commands into the computer, so that you don't have to type in all binary. There are other types of shell, such as `csh`, which look slightly different and have slightly different commands.

One way to tell if you are running `bash` or `csh` is to look at the command prompt. In the figure above, you are presented with a dollar sign, which usually indicates the bash environment. The csh environment uses a percentage sign as a command prompt. This tutorial will focus on the `bash` environment, so if you see the percentage sign, type the word `bash` into the terminal and hit enter.

2 Your First Commands

Like any operating system, the UNIX environment has files located in a structured hierarchy of directories (equivalent to folders in Windows or Mac OS). Without a GUI, to navigate this hierarchy you must use commands to move from directory to directory, to list the contents of a directory, and to create, move, copy, and delete files. In this section you will learn some of the most basic commands to help you navigate the UNIX environment.

Note for Cygwin users: Because of the way Cygwin operates from within Windows, your results for the next few sections may vary. Ask one of us if you are confused!

2.1 Moving around the system

—pwd

One of the most basic commands answers one of the most basic questions: "Where are we?" The `pwd` command **P**rints the **W**orking **D**irectory, the one you're currently in.

```
CBG002827:~ mjohnson$ pwd
/Users/mjohnson
```

Most likely, you are currently in your *home directory*, which is the default location whenever you login to the terminal. The highest (or deepest) directory in any file system is the root

directory, indicated in UNIX by `/` (a forward slash). The directory listed above is therefore three tiers into the hierarchy: root, then "Users," and then the home directory for the user "mjohnson."

—ls

In order to further navigate through the file system, you will first need to know where you can go. You can **ls**t the files and directories within the current directory with the command `ls`:

```
CBG002827:~ mjohnson$ ls
Desktop    Downloads  Library    Music      Pictures
Documents  Dropbox    Movies   Public
```

This is a pretty typical output for MacOS users, your results may vary. In particular, if you are logging onto a UNIX server for the first time, you may get no result at all! This is okay, you will soon learn how to fill your home directory.

It is possible to list the files in other directories besides the current one. To do this, you must refer to the directory you want to look in. For example:

```
CBG002827:~ mjohnson$ ls /
Applications     Users            etc              private
Library          Volumes          home             sbin
Network          bin              mach_kernel      tmp
System           cores            net              usr
User Information dev              opt              var
```

Again, your results may vary. In this example, you have *passed an argument* to the command `ls`. The argument is the forward slash, representing the *root directory*. Given the results of this example, you could pass other directory names to `ls`, such as: `/usr` or `/opt`. Try it!

—mkdir

To **m**a**k**e a new **dir**ectory, you use the command `mkdir` along with the name of the new directory:

```
CBG002827:~ mjohnson$ mkdir myfirstdir
CBG002827:~ mjohnson$
```

Notice that there was no output at all from the `mkdir` command. This is normal! The command will only produce output if there is an error; for example, if you tried to create a directory that already exists.

To access files in this new directory, you will have to **c**hange **d**irectories with the `cd`

command:

—cd

```
CBG002827:~ mjohnson$ pwd
/Users/mjohnson
CBG002827:~ mjohnson$ cd myfirstdir
CBG002827:myfirstdir mjohnson$ pwd
/Users/mjohnson/myfirstdir
CBG002827:myfirstdir mjohnson$
```

Once again there is no output of a successful `cd` command, but you can check that you have successfully changed directories with `pwd`. You can also see that the command prompt changed to indicate what directory you are in.

In the case above, you used `cd myfirstdir` to change to a directory located within the current directory. This is referred to as a *relative path* because you are referring to the location of *myfirstdir* relative to where you are currently.

Alternatively, you could refer to the *full path*, which above would be `cd /Users/mjohnson/myfirstdir` . Your exact path may vary, but using to the full path would ensure that you are referring to the correct location.

As you are navigating throughout the file system, there are several useful shortcuts built into every UNIX system. Most systems come equipped with *tab completion*, which means you can type half of a command, hit the "Tab" key and the system will try to complete the command for you. In addition, you can use the up and down arrow keys to scroll through commands you have previously entered.

The current directory is denoted as a single period: `.` , and the directory directly above the current directory is denoted as two periods: `..`

Additionally, `~/` can be used as a shortcut for your home directory in the UNIX terminal. You can use any of these with most commands, including `cd` and `ls`.

```
CBG002827:myfirstdir mjohnson$ pwd
/Users/mjohnson/myfirstdir
CBG002827:myfirstdir mjohnson$ ls .
CBG002827:myfirstdir mjohnson$ ls ..
Desktop    Downloads  Library    Music      Pictures
Documents  Dropbox    Movies      Public    myfirstdir
CBG002827:myfirstdir mjohnson$ cd ..
CBG002827:~ mjohnson$ ls .
Desktop    Downloads  Library    Music      Pictures
Documents  Dropbox    Movies      Public    myfirstdir
CBG002827:~ mjohnson$ cd myfirstdir/
CBG002827:myfirstdir mjohnson$ cd ~/
CBG002827:~ mjohnson$
```

Using `ls`, `cd`, and the directory shortcuts, explore the file system! How many files do you think there are in the `/usr/bin` directory?

2.2 Creating, editing, and removing files

—cat

Now that you are experienced with the directory structure, it is time to fill those directories with files! The simplest way to create a file from the command line is with `cat`. This command is also used to con**cat**enate multiple files, but is also used to add text to files. In the most basic usage, without any arguments, cat will simply repeat all text typed.

To give it a try, type the `cat` command and hit enter. You will be presented with a blank line with no command prompt. Type something, such as `Hello world!` and hit enter, and it will be repeated back to you!

```
CBG002827:usr mjohnson$ cd ~/myfirstdir/
CBG002827:myfirstdir mjohnson$ cat
Hello World!
Hello World!
^C
CBG002827:myfirstdir mjohnson$
```

To return the command prompt, press Ctrl and C at the same time, shown above as `^C`. (In general, this is a good way to cancel commands.)

The reason `cat` is printing back to the screen is that it is the **st**an**d**ard **out**put for the command, or *stdout*. It is often useful for programs to output to stdout, so the user may decide what to do with the output. In UNIX, the output can be *redirected* using the greater-than symbol. Think of it like an arrow, so that anything that the program would normally print on the screen would instead go to a different destination, indicated by an arrow. If you try this with `cat`, you can specify a file and all output will go there.

```
CBG002827:myfirstdir mjohnson$ cat junk.txt
cat: junk.txt: No such file or directory
CBG002827:myfirstdir mjohnson$ cat > junk.txt
This will be output to junk.txt, not the screen.
^C
CBG002827:myfirstdir mjohnson$ cat junk.txt
This will be output to junk.txt, not the screen.
CBG002827:myfirstdir mjohnson$
```

The first command demonstrates that the file junk.txt does not exist, yet. Next, you open the cat program but with the instruction to *redirect* the output to the file junk.txt. This time, when you type within `cat`, the output does not go to the screen, but into the file

instead. The final command with `cat` prints the contents of the file junk.txt, containing your output! Again, use Ctrl-C to exit `cat`.

—**cp** and —**mv**

Managing your files involves copious use of the commands `mv` and `cp`. These commands move and copy files, respectively. The general usage of both commands is `cp source destination`. The `mv` command usually moves files to a different directory, but you can also use `mv` to rename a file in the current directory.

```
CBG002827:myfirstdir mjohnson$ ls
junk.txt
CBG002827:myfirstdir mjohnson$ cp junk.txt junk2.txt
CBG002827:myfirstdir mjohnson$ ls
junk.txt  junk2.txt
CBG002827:myfirstdir mjohnson$ mv junk2.txt junk3.txt
CBG002827:myfirstdir mjohnson$ ls
junk.txt  junk3.txt
CBG002827:myfirstdir mjohnson$ mkdir junkdir
CBG002827:myfirstdir mjohnson$ mv junk3.txt junkdir
CBG002827:myfirstdir mjohnson$ ls
junk.txt junkdir
CBG002827:myfirstdir mjohnson$ ls junkdir
junk3.txt
CBG002827:myfirstdir mjohnson$
```

In the second `mv` command above, the system recognizes that *junkdir* is a directory. A file cannot overwrite a directory, so instead, `mv` places the file *junk3.txt* into the *junkdir* directory.

—**rm**

You can remove files with the `rm` command. To remove a file, simply list its name. You do not have to be in a directory to delete files within it, simply specify where the file is.

```
CBG002827:myfirstdir mjohnson$ ls junkdir
junk3.txt
CBG002827:myfirstdir mjohnson$ rm junkdir/junk3.txt
CBG002827:myfirstdir mjohnson$ ls junkdir
CBG002827:myfirstdir mjohnson$
```

And just like that, junk3.txt is gone! **BE VERY CAREFUL WITH rm!** There is no "undo" in UNIX; once a file has been removed, it is usually gone forever, unless you have backups. For your protection, most important system files cannot be removed simply with `rm`.

2.3 Command Options

Many commands have different options which alter their default behavior. When you execute a command, you can specify these different options using *flags*, which begin with either one or two dashes. The `ls` command has several useful options controlling its output.

```
CBG002827:myfirstdir mjohnson$ ls
junk.txt junkdir
CBG002827:myfirstdir mjohnson$ ls -a
.         ..        junk.txt junkdir
CBG002827:myfirstdir mjohnson$ ls -t
junkdir  junk.txt
CBG002827:myfirstdir mjohnson$ ls -l
total 8
-rw-r--r--  1 mjohnson  staff  65 Sep 10 10:55 junk.txt
drwxr-xr-x  2 mjohnson  staff  68 Sep 10 11:20 junkdir
CBG002827:myfirstdir mjohnson$
```

`ls -a` lists all of the files, including hidden files, that are not normally listed with `ls`. In this case, it has added the current and next highest directories, `.` and `..`

`ls -t` lists the file in order of most recently modified. The directory "junkdir" was modified more recently than the file "junk.txt"

`ls -l` lists the files in "list mode," where a lot more information is available, including the Permissions, date modified, and the file sizes.

For many UNIX commands, you can chain together command line options into one command. For example, `ls -h` is useful in combination with `ls -l`, for listing file sizes in "human readable format," so it will read 1K rather than 1024. In fact, you can combine all four of these flags into one command:

```
CBG002827:myfirstdir mjohnson$ ls -lath
total 8
drwxr-xr-x   2 mjohnson  staff    68B Sep 10 11:20 junkdir
drwxr-xr-x   4 mjohnson  staff   136B Sep 10 11:14 .
-rw-r--r--   1 mjohnson  staff    65B Sep 10 10:55 junk.txt
drwxr-xr-x+ 29 mjohnson  staff   986B Sep 10 09:26 ..
```

Here, all files are listed in "list mode," sorted by most recently modified and with file sizes in human readable format.

Another useful command line flag is the *recursive* option, `-r` . This applies a command to all directories and subdirectories. You must use `cp -r` to copy a directory (and everything within it), and `rm -r` to remove a directory (and everything within it).

—man

To find out what flags are possible for a command, use the **man**ual command `man`. If you type `man ls` and hit enter, a screen will appear telling you details about the `ls` command, including options. Use the arrow keys or spacebar to scroll through the document. When you are done viewing the document, press `q` to exit.

Other, non UNIX programs may have different help systems. Sometimes, a program will return the help information by default if you do not enter any command line flags. Alternatively, you can sometimes get help with `program -h` or `program --help`.

3 Variables

In programming, as with mathematics, variables are standins for other information. In UNIX, they are frequently used as shortcuts for very long items you may not want to type into the command line every time. For instance, imagine that you want to copy a file deep within a UNIX directory system, to a location such as /Users/mjohnson/data/projectname/temporary/jobstatus That's a lot to type! Instead you can use a variable to save you keystrokes:

```
CBG002827:~ mjohnson$ mkdir -p data/projectname/temporary/jobs
tatus
CBG002827:~ mjohnson$ SHORTCUT=/Users/mjohnson/data/project
name/temporary/jobstatus
CBG002827:~ mjohnson$ cd $SHORTCUT
CBG002827:jobstatus mjohnson$ pwd
/Users/mjohnson/data/projectname/temporary/jobstatus
CBG002827:jobstatus mjohnson$
```

Here, `mkdir -p` creates all the necessary subdirectories if they do not exist. It is customary in UNIX to use capital letters to denote variables, so they are distinct from other information. Do not use spaces before or after the equals sign when assigning a variable in bash, or it will not work!

You then reference the variable in other commands using the $

NOTE: This method of creating a variable is specific to the bash environment, and will not work in other shells.

—echo

In order to figure out what a variable currently represents, use the `echo` command. It can also be used to print out strings to the screen.

```
CBG002827:~ mjohnson$ echo $SHORTCUT
/Users/mjohnson/data/projectname/temporary/jobstatus
CBG002827:~ mjohnson$
```

Variables do not just stand in for long strings you don't want to type out. In programming, they are used to stand in for a value that might change each time you run a program. A simple example follows, using `echo`.

```
CBG002827:~ mjohnson$ FIZZYDRINK=soda
CBG002827:~ mjohnson$ REGION=Northeast
CBG002827:~ mjohnson$ echo "In the $REGION, it's called $FIZZYDRINK."
In the Northeast, it's called soda.
CBG002827:~ mjohnson$ FIZZYDRINK=coke
CBG002827:~ mjohnson$ REGION=South
CBG002827:~ mjohnson$ echo "In the $REGION, it's called $FIZZYDRINK."
In the South, it's called coke
CBG002827:~ mjohnson$
```

In this way you could keep repeating the `echo` command but with different values for the variables `$FIZZYDRINK` and `$REGION`.

—$PATH

In UNIX, some variables are special, they have a set name in the system, and you should change them only if you know what you are doing. These are referred to as *environmental variables*, and `$PATH` is one of them.

Every command you have entered so far references code saved somewhere in the system. To see where a command is being executed, use the `which` command:

```
CBG002827:~ mjohnson$ which ls
/bin/ls
CBG002827:~ mjohnson$ which echo
/bin/echo
CBG002827:~ mjohnson$ which man
/usr/bin/man
CBG002827:~ mjohnson$
```

How does the system know to find `ls` and `echo` in one location, but `man` in another? The system calls the environmental variable `$PATH`. You can view `$PATH` on your system with `echo $PATH`. The result will be a set of directories separated by colons. When you type a command, the system will search the directories, in this order, for a file matching that name, and execute it.

You can change the `$PATH` on your system much in the same way you change other variables. The difference is you must tell the system you are updating an environmental variable.

```
CBG002827:~ mjohnson$ echo $PATH
/opt/local/bin:/opt/local/sbin:/usr/bin:
/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/texbin:
/Users/mjohnson/bin
CBG002827:~ mjohnson$ export PATH=/Users/mjohnson/myfirstdir/:$PATH
```

```
CBG002827:~ mjohnson$ echo $PATH
/Users/mjohnson/myfirstdir/:/opt/local/bin:
/opt/local/sbin:/usr/bin:/bin:/usr/sbin:/sbin:
/usr/local/bin:/usr/texbin:/Users/mjohnson/bin
CBG002827:~ mjohnson$
```

Here, the `export` command adds a new directory to the beginning of the `$PATH`. Now, the system will first search for commands in `~/myfirstdir` before moving on to the other locations.

4 Viewing Files

There are a number of UNIX tools for viewing and/or editing files, each of which has its uses.

—less

There are two simple text viewing utilities installed in UNIX systems, `more` and `less`. The `more` command will read an entire text file and allow you to scroll forwards only (with the arrow keys). By contrast, `less` is much more versatile, allowing you to scroll in both directions. It also only reads in part of the file at a time, a useful feature for bioinformatics files that can get quite large. (Yes, the name of `less` is a pun, as in *less is more*.) Much like the `man` command, you exit out of `less` and `more` by pressing `q`.

—nano

There are several command line text editors, such as `vi`, `emacs`, and `nano`. Much like the Python/Perl/Ruby debate about scripting languages, the choice of text editor can also cause lengthy discussion among too-enthusiastic programmers. All that is important for beginners is to pick one of them, and this author is used to `nano`. Text editors with a GUI (such as TextWrangler or Notepad++) are also useful. However, sometimes it is easier and faster to simply edit the file without opening an additional program.

Navigate to the directory containing the junk.txt file you created with `cat` and enter the command: `nano junk.txt`

```
  GNU nano 2.0.6              File: junk.txt

This will not be output to the screen, but to the file junk.txt.
```

```
                                [ Read 1 line ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Within `nano`, you can move the cursor around with the arrow keys and start editing the file. As soon as you enter something new, "Modified" will appear in the upper right corner. After you enter some new text, it is time to quit and save the file. In `nano`, you quit by pressing the Control and X keys at the same time. This is listed at the bottom of the screen as `^X` Since the file has been modified, it will ask you if you want to save your changes. Type `y` for yes and `n` for no. If you choose to accept your changes, `nano` will then ask how to name the file. To overwrite the previous file, simply hit enter, and `nano` will exit back to the UNIX prompt.

—**head** and —**tail**

These two commands are useful for getting a quick look at a file, and perhaps using it for other purposes. By default, they both print ten lines of a specified text file; `head` will print the first ten lines, while `tail` will print the final ten. You can specify other numbers of lines with the -n flag, such as the first 100 lines with: `head -n 100 myfile.txt`

Both `head` and `tail` print to *stdout*, so their output can be *redirected* to another file.

```
CBG002827:~ mjohnson$ head /usr/share/dict/words
A
a
aa
aal
aalii
aam
Aani
aardvark
aardwolf
Aaron
CBG002827:~ mjohnson$ tail /usr/share/dict/words
zymotoxic
zymurgy
Zyrenian
Zyrian
Zyryan
zythem
Zythia
zythum
Zyzomys
Zyzzogeton
CBG002827:~ mjohnson$ tail -n 100 /usr/share/dict/words >first100words.txt
```

```
CBG002827:~ mjohnson$
```

Some useful applications of `head` include:

- Reading the comments at the beginning of a README file.
- Printing the first 10 sequences from a FASTA file, to create a "test file" for testing your bioinformatics pipeline.
- *Advanced* Searching a file with `grep` and only viewing the first result.

Some useful applications of `tail` include:

- Checking the status of a long-running job by viewing the end of the log file.
- Grabbing the most recent phylogenetic tree from a Bayesian analysis.
- *Advanced* Monitoring a log file in real-time with `tail -f logfile.txt`

5 Flow Control

Before you can advance to the scripting section, it is useful to know a bit more about *flow control* in UNIX commands. You already have learned about *redirecting* output from stdout to a file with >. Other commands will allow you to append to existing files, or to direct the output of one command into a second command.

5.1 Appending

You can append new text to a file using two greater-than symbols:

```
CBG002827:myfirstdir mjohnson$ cat junk.txt
This will not be output to the screen, but to the file junk.txt.
CBG002827:myfirstdir mjohnson$ cat >>junk.txt
I am adding text to the file!
^C
CBG002827:myfirstdir mjohnson$ cat junk.txt
This will not be output to the screen, but to the file junk.txt.
I am adding text to the file!
CBG002827:myfirstdir mjohnson$
```

Note that if you use only one greater than sign (redirect output) rather than two (append), it would *overwrite* the contents of junk.txt.

5.2 stdin

Much like *stdout*, many programs can read in data from *stdin*, rather than reading a file. To do this, you use the less-than symbol, which can be used as if you are pointing from a file into a command:

```
CBG002827:myfirstdir mjohnson$ cat junk.txt
This will not be output to the screen, but to the file junk.txt.
I am adding text to the file!
CBG002827:myfirstdir mjohnson$ cat < junk.txt
This will not be output to the screen, but to the file junk.txt.
I am adding text to the file!
CBG002827:myfirstdir mjohnson$
```

So you can see that *stdin* by itself is not too useful, as it is (usually) functionally equivalent to simply reading from a file.

—read

A more advanced example of using *stdin* uses the `read` command to assign variables. For example, say you want to store the size of a directory as a variable. The size of a directory is listed at the beginning when using `ls -l`. If you save the `ls -l` results to a file, `read` can read the first line of that file and assign its contents to variables.

```
CBG002827:~ mjohnson$ ls -l /usr/bin > usrbin.txt
CBG002827:~ mjohnson$ head -n 3 usrbin.txt
total 137720
-rwxr-xr-x   6 root   wheel       925 Mar 12  2013 2to3
-rwxr-xr-x   6 root   wheel       925 Mar 12  2013 2to3-
CBG002827:~ mjohnson$ read TOTAL SIZE < usrbin.txt
CBG002827:~ mjohnson$ echo $TOTAL
total
CBG002827:~ mjohnson$ echo $SIZE
137720
CBG002827:~ mjohnson$
```

When `read` loads the first line of `usrbin.txt`, it stores the first value "total" in the variable `$TOTAL` and the second value "137720" in the variable `$SIZE`.

Additionally, *stdin* is used frequently when *piping* the output of one program into another, as demonstrated in the next section.

5.3 Piping

In UNIX, a *pipe* operates to send the output of one program into the input of another program. To demonstrate how piping works, consider the command `wc`.

—wc

On its own, `wc` acts like `cat`; you can type as much as you like, then use Control-D to exit. The command will then show you the number of lines, number of words, and number of characters in the text you entered. You can also specify a file for `wc` to count. If you only want `wc` to return the number of lines, you can use the `-l` flag.

```
CBG002827:myfirstdir mjohnson$ wc
The quick brown fox jumped over the lazy dog.
       1       9      46
CBG002827:myfirstdir mjohnson$ wc junk.txt
       2      20      95 junk.txt
CBG002827:myfirstdir mjohnson$ wc -l junk.txt
       2 junk.txt
CBG002827:myfirstdir mjohnson$
```

One use of wc is to keep track of the number of files in a certain directory. Without piping, you could do this in two steps.

```
CBG002827:myfirstdir mjohnson$ ls -l /usr/bin > usrbin.txt
CBG002827:myfirstdir mjohnson$ wc -l < usrbin.txt
    1088
CBG002827:myfirstdir mjohnson$
```

Your results may vary. Note that `ls -l` will insert a line at the beginning describing the size of the directory, so there are actually 1,087 files in `/usr/bin`. In order to count the number of files, you have to create an intermediate file (usrbin.txt). You write out to that file from `ls`, and read in from that file to `wc`. You can actually do both in one command by using a pipe: `|`

```
CBG002827:myfirstdir mjohnson$ ls -l /usr/bin | wc -l
    1088
CBG002827:myfirstdir mjohnson$
```

Here, you are telling `wc` to use the output of the command to the left as its input. Any commands that can read from *stdin* and write to *stdout* can use piping. For example, you could also pipe the output of `ls` to `less` so you can scroll through the file names.

6 Scripting in bash

Now that you are familiar with a number of UNIX commands, the next step is to join the commands together into a *script*. In `bash`, typically a script is a text file containing a set of commands you wish to execute. This is very useful when you need to execute the same commands multiple times.

Consider the following script, which executes several of the commands you've seen from earlier sections.

```
#How many words are in the dictionary?
cd ~/
mkdir fromscript
cd fromscript
DICT=/usr/share/dict/words
wc -l $DICT > wordsindict.txt           #Count the words in a dictionary.
cat wordsindict.txt
read LINES DICT < wordsindict.txt
echo "There are $LINES words in the dictionary at $DICT"
```

Notice that there is no command prompt here.

Notice that the words in italics are preceded by a pound sign (#). These are *comments*, and are not read by `bash` when executing the script. Every programming language has a way to insert comments, which are useful if you or someone else reads your script later.

Instead, these commands can be saved into a text file, such as `firstbash.sh`. You then execute the contents of the text file with the command `bash`.

```
CBG002827:~ mjohnson$ bash firstbash.sh
  235886 /usr/share/dict/words
There are 235886 words in the dictionary at /usr/share/dict/words
CBG002827:~ mjohnson$
```

One of the key ideas in designing any script is knowing what kind of data you are dealing with. Here, we know that:

- a dictionary of words exists at `/usr/share/dict/words`
- the output of `wc -l` has two pieces of information: the number of lines, and the source.

Therefore, we can plan accordingly, and store the values in the correct variables using `read`. If the location of the dictionary changed, we could change the value accordingly in the script file.

If you run the script a second time, you should get a third line of output. This indicates that `mkdir` encountered an error, that the directory `~/fromscript/` already exists.

6.1 Permissions

Every file and directory in a UNIX system has *permissions* that indicate which users are able to read, edit, or execute the file or directory. This is most useful on a shared computer

server, where a systems administrator can make some files useable by all or only certain groups of users.

To view the permissions for files in a directory, use `ls -la`

```
CBG002827:~ mjohnson$ ls -la
total 256
drwxr-xr-x+  34 mjohnson  staff   1156 Sep 11 12:33 .
drwxr-xr-x    8 root      admin    272 Aug 26 10:19 ..
-rw-r--r--    1 mjohnson  staff   1048 Sep 11 10:08 first100words.txt
-rw-r--r--   1 mjohnson  staff    216 Sep 11 12:56 firstbash.sh
drwxr-xr-x    3 mjohnson  staff    102 Sep 11 12:24 fromscript
drwxr-xr-x    6 mjohnson  staff    204 Sep 11 12:17 myfirstdir
-rw-r--r--    1 mjohnson  staff  72223 Sep 11 12:37 usrbin.txt
CBG002827:~ mjohnson$
```

The string of text on the left of each entry, such as `-rw-r--r--` indicates the *permissions* status for everything in this directory. There are ten entries for each item:

- Directories are noted in the first position with a `d`.
- The next three positions show the permissions for the owner of the file/directory (shown in the third column)

  r indicates that the file is readable

  w indicates that the file is writable (editable). For directories it means that new files can be added by this user.

  x indicates that the file is executable. For directories it means that the user can `cd` to this directory.
- The next three positions show the read/write/execute permissions for the group the owner belongs to (shown in the fourth column)
- The final three positions show the read/write/execute permissions for everyone else.

You can see that your script `firstbash.sh` is not listed as executable. To execute commands in the current directory, you need to use: `./command`.

```
CBG002827:~ mjohnson$ ./firstbash.sh
-bash: ./firstbash.sh: Permission denied
CBG002827:~ mjohnson$
```

This is unsuccessful because your script does not have the correct permissions. In order to run your script `firstbash.sh` as if it were a normal UNIX command, you must do two things:

- Change the permissions on the file to make it executable

- Add a *shebang* to your file so the system knows how to interpret it.

—chmod

The `chmod` command changes the permissions of files and directories. To simply give yourself permission to execute a script, use `chmod +x firstbash.sh`. If you are on a computer cluster and want all users to be able to execute your script, use `chmod a+x`. Other permissions permutations are possible by changing the `x` to `r` for read permissions or `w` for write (edit) permissions. You can also change the plus to a minus to remove permissions.

Be careful, though, as it is possible to remove your own ability to access files and directories!

Changing the permissions for the script `firstbash.sh` will probably be enough to allow it to be executed with `./firstbash.sh`

However, the proper way to tell the system how to run your script is to add the *shebang* line to the beginning of the script. The *shebang* is a nickname for `#!` and tells the system "What follows is instructions on how to interpret this code." You may end up writing scripts using other shell languages, or Perl, or Python. Here, we will want to tell the system to use `bash` and not another interpreter.

Add the following line to the beginning of `firstscript.sh` (for example, using `nano`):

```
#!/bin/bash
```

Save the file, and now the system will always know what to do with your script.

7 Additional resources

The most important advice for any beginning programer is: *Use Google.* Many problems are encountered over and over by beginners and experts alike, and the internet is full of people asking and answering programming questions. If you need to narrow your search, add "bash" or "unix" to the search terms. Using Google to search for errors is particularly useful.

Frequently, Googles searches for programming questions end up at Stack Overflow, an excellent community for help at all levels.
`http://stackoverflow.com`

If you want more information about scripting in `bash`, there is an excellent in-depth tutorial here: `http://mywiki.wooledge.org/BashGuide`

An excellent "cheat sheet" of UNIX commands can be found here:
`http://files.fosswire.com/2007/08/fwunixref.pdf`