

UNIX - Command-Line Survival Guide

Files, directories, commands, text editors

Simon Prochnik & Lincoln Stein

Lecture Notes

- [What is the Command Line?](#)
 - [Logging In](#)
 - [The Desktop](#)
 - [The Shell](#)
 - [Home Sweet Home](#)
 - [Getting Around](#)
 - [Running Commands](#)
 - [Command Redirection](#)
 - [Pipes](#)
-

What is the Command Line?

Underlying the pretty Mac OSX GUI is a powerful command-line operating system. The command line gives you access to the internals of the OS, and is also a convenient way to write custom software and scripts.

Many bioinformatics tools are written to run on the command line and have no graphical interface. In many cases, a command line tool is more versatile than a graphical tool, because you can easily combine command line tools into automated scripts that accomplish tasks without human intervention.

In this course, we will be writing Perl scripts that are completely command-line based.

Logging into Your Workstation

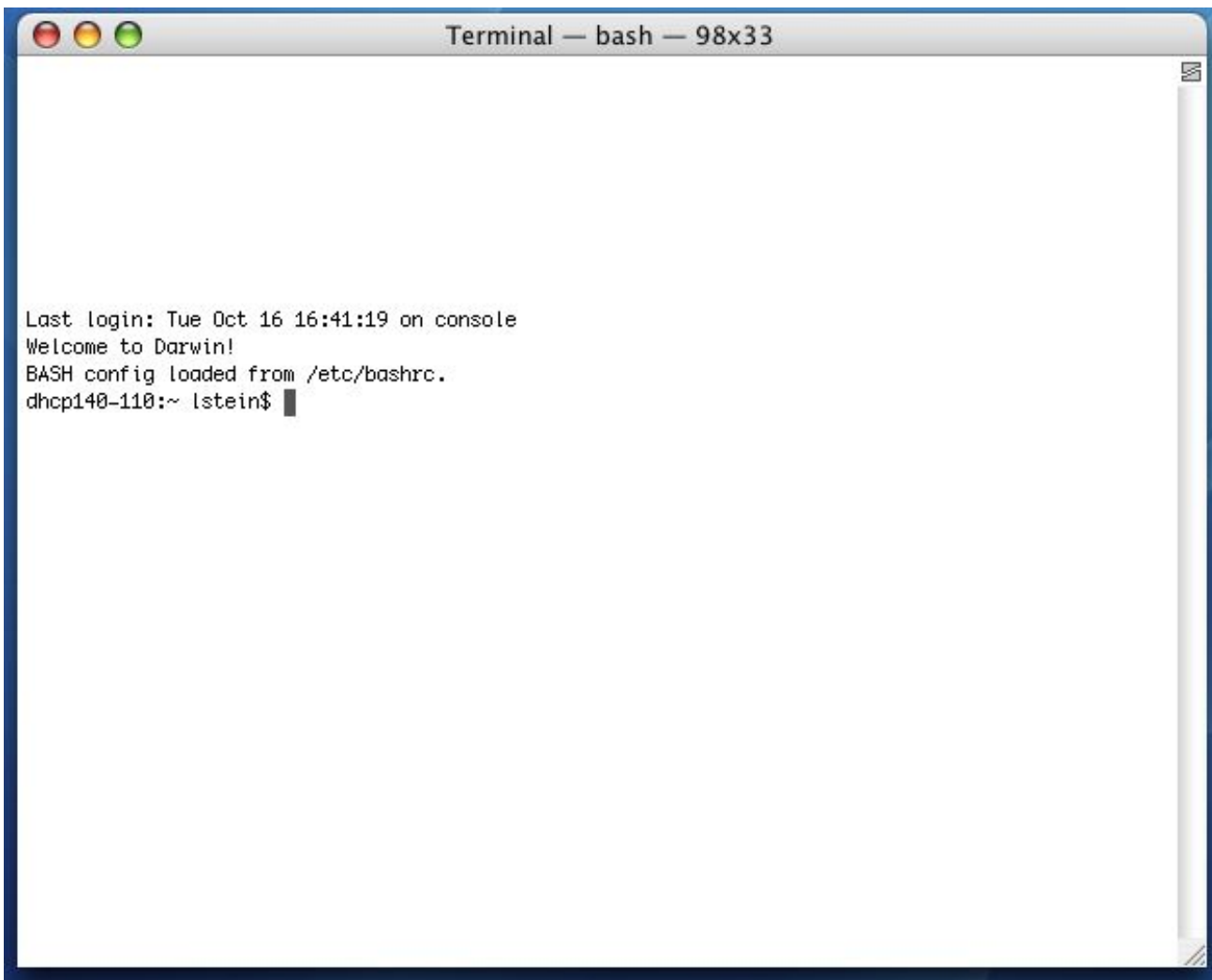
Your workstation is an iMac. To log into it, provide the following information:

Your username: the initial of your first name, followed by your full last name. For example, my username is **srobb** for **sofia robb**

Your password: **changeme**

Bringing up the Command Line

To bring up the command line, use the Finder to navigate to *Applications->Utilities* and double-click on the *Terminal* application. This will bring up a window like the following:

A screenshot of an OSX Terminal window. The title bar at the top reads "Terminal — bash — 98x33". The window contains the following text:

```
Last login: Tue Oct 16 16:41:19 on console
Welcome to Darwin!
BASH config loaded from /etc/bashrc.
dhcp140-110:~ lstein$ █
```

OSX Terminal

You will be using this application a lot, so I suggest that you drag the Terminal icon into the shortcuts bar at the bottom of your screen.

OK. I've Logged in. What Now?

The terminal window is running a **shell** called "bash." The shell is a loop that:

1. Prints a prompt
2. Reads a line of input from the keyboard
3. Parses the line into one or more commands
4. Executes the commands (which usually print some output to the terminal)
5. Go back 1.

There are many different shells with bizarre names like **bash**, **sh**, **csh**, **tcsh**, **ksh**, and **zsh**. The "sh" part means shell. Each shell was designed for the purpose of confusing you and tripping you up. We have set up your accounts to use **bash**. Stay with **bash** and you'll get used to it, eventually.

Command-Line Prompt

Most of bioinformatics is done with command-line software, so you should take some time to learn to use the shell effectively.

This is a command line prompt:

```
bush202>
```

This is another:

```
(~) 51%
```

This is another:

```
srobb@bush202 1:12PM>
```

What you get depends on how the system administrator has customized your login. You can customize yourself when you know how.

The prompt tells you the shell is ready to accept a command. When a long-running command is going, the prompt will not reappear until the system is ready to deal with your next request.

Issuing Commands

Type in a command and press the <Enter> key. If the command has output, it will appear on the screen. Example:

```
(~) 53% ls -F
GNUstep/          cool_elegans.movies.txt  man/
INBOX             docs/                   mtv/
INBOX~           etc/                    nsmail/
Mail@            games/                  pcod/
News/            get_this_book.txt       projects/
axhome/          jcod/                   public_html/
bin/             lib/                    src/
build/           linux/                  tmp/
ccod/
(~) 54%
```

The command here is `ls -F`, which produces a listing of files and directories in the current directory (more on which later). After its output, the command prompt appears again.

Some programs will take a long time to run. After you issue their command name, you won't recover the shell prompt until they're done. You can either launch a new shell (from Terminal's File menu), or run the command in the background using the ampersand:

```
(~) 54% long_running_application&
(~) 55%
```

The command will now run in the background until it is finished. If it has any output, the output will be printed to the terminal window. You may wish to redirect the output as described later.

Command Line Editing

Most shells offer command line editing. Up until the moment you press <Enter>, you can go back over the command line and edit it using the keyboard. Here are the most useful keystrokes:

Backspace

Delete the previous character and back up one.

Left arrow, right arrow

Move the text insertion point (cursor) one character to the left or right.

control-A (^A)

Move the cursor to the beginning of the line. Mnemonic: A is first letter of alphabet

control-E (^E)

Move the cursor to the end of the line. Mnemonic: <E> for the End (^Z was already taken for something else).

control-D (^D)

Delete the character currently under the cursor. D=Delete.

control-K (^K)

Delete the entire line from the cursor to the end. K=Kill. The line isn't actually deleted, but put into a temporary holding place called the "kill buffer".

control-Y (^Y)

Paste the contents of the kill buffer onto the command line starting at the cursor. Y=Yank.

Up arrow, down arrow

Move up and down in the command history. This lets you reissue previous commands, possibly after modifying them.

There are also some useful shell commands you can issue:

history

Show all the commands that you have issued recently, nicely numbered.

!*<number>*

Reissue an old command, based on its number (which you can get from *history*)

!!

Reissue the immediate previous command.

!*<partial command string>*

Reissue the previous command that began with the indicated letters. For example *!!* would reissue the *ls -F* command from the earlier example.

bash offers automatic command completion and spelling correction. If you type part of a command and then the tab key, it will prompt you with all the possible completions of the command. For example:

```
(~) 51% fd<tab>
```

```
(~) 51% fd
```

```
fd2ps    fdesign  fdformat fdlist   fdmount  fdmountd fdrawcmd fdumount
```

```
(~) 51%
```

If you hit tab after typing a command, but before pressing <Enter>, **bash** will prompt you with a list of file names. This is because many commands operate on files.

Wildcards

You can use wildcards when referring to files. "*" refers to zero or more characters. "?" refers to any single character. For example, to list all files with the extension ".txt", run **ls** with the pattern "*.txt":

```
(~) 56% ls -F *.txt
```

```
final_exam_questions.txt  genomics_problem.txt
```

```
genebridge.txt             mapping_run.txt
```

There are several more advanced types of wildcard patterns which you can read about in the **tcsh** manual page. For example, you can refer to files beginning with the characters "f" or "g" and ending with ".txt" this way:

```
(~) 57% ls -F [f-g]*.txt
```

```
final_exam_questions.txt  genebridge.txt
```

```
genomics_problem.txt
```

Home Sweet Home

When you first log in, you'll be placed in a part of the system that is your personal domain, called the *home directory*. You are free to do with this area what you will: in particular you can create and delete files and other directories. In general, you cannot create files elsewhere in the system.

Your home directory lives somewhere way down deep in the bowels of the system. On our iMacs, it is a directory with the same name as your login name, located in **/Users**. The full directory path is therefore **/Users/username**. Since this is a pain to write, the shell allows you to abbreviate it as `~username` (where "username" is your user name), or simply as `~`. The weird character (technically called the "twiddle") is usually hidden at the upper left corner of your keyboard.

To see what is in your home directory, issue the command `ls -F`:

```
(~) % ls -F
INBOX          Mail/          News/          nsmail/       public_html/
```

This shows one file "INBOX" and four directories ("Mail", "News") and so on. (The "-F" in the command turns on fancy mode, which appends special characters to directory listings to tell you more about what you're seeing. "/" means directory.)

In addition to the files and directories shown with `ls -F`, there may be one or more hidden files. These are files and directories whose names start with a "." (technically called the "dot" character). To see these hidden files, add an "a" to the options sent to the `ls` command:

```
(~) % ls -aF
./              .cshrc         .login         Mail/
../            .fetchhost    .netscape/   News/
.Xauthority    .fvwmrc       .xinitrc*     nsmail/
.Xdefaults     .history      .xsession@    public_html/
.bash_profile  .less         .xsession-errors
.bashrc        .lessrc       INBOX
```

Whoa! There's a lot of hidden stuff there. But don't go deleting dot files willy-nilly. Many of them are essential configuration files for commands and other programs. For example, the *.profile* file contains configuration information for the **bash** shell. You can peek into it and see all of **bash**'s many options. You can edit it (when you know what you're doing) in order to change things like the command prompt and command search path.

Getting Around

You can move around from directory to directory using the `cd` command. Give the name of the directory you want to move to, or give no name to move back to your home directory. Use the `pwd` command to see where you are (or rely on the prompt, if configured):

```
(~/docs/grad_course/i) 56% cd
(~) 57% cd /
(/) 58% ls -F
bin/          docsc/        gmon.out      mnt/          sbin/
```

```

boot/          etc/          home@         net/          tmp/
cdrom/         fastboot     lib/          proc/         usr/
dev/           floppy/      lost+found/   root/         var/
(/) 59% cd ~/docs/
(~/docs) 60% pwd
/usr/home/lstein/docs
(~/docs) 62% cd ../projects/
(~/projects) 63% ls
Ace-browser/   bass.patch
Ace-perl/      cgi/
Foo/           cgi3/
Interface/     computertalk/
Net-Interface-0.02/ crypt-cbc.patch
Net-Interface-0.02.tar.gz fixer/
Pts/           fixer.tcsh
Pts.bak/       introspect.pl*
PubMed/        introspection.pm
SNPdb/         rhmap/
Tie-DBI/       sbox/
ace/           sbox-1.00/
atir/          sbox-1.00.tgz
bass-1.30a/    zhmapper.tar.gz
bass-1.30a.tar.gz
(~/projects) 64%

```

Each directory contains two special hidden directories named "." and "..". "." refers always to the directory in which it is located. ".." refers always to the parent of the directory. This lets you move upward in the directory hierarchy like this:

```
(~/docs) 64% cd ..
```

and to do arbitrarily weird things like this:

```
(~/docs) 65% cd ../../docs
```

The latter command moves upward to levels, and then into a directory named "docs".

If you get lost, the *pwd* command prints out the full path to the current directory:

```
(~) 56% pwd
/Users/lstein
```

Essential Unix Commands

With the exception of a few commands that are built directly into the shell, all Unix commands are standalone executable programs. When you type the name of a command, the shell will search through all the directories listed in the PATH environment variable for an executable of the same name. If found, the shell will execute the command.

Otherwise, it will give a "command not found" error.

Most commands live in `/bin`, `/usr/bin`, or `/usr/local/bin`.

Getting Information About Commands

The **man** command will give a brief synopsis of the command:

```
(~) 76% man wc
Formatting page, please wait...
WC(1)                                                    WC(1)

NAME
    wc - print the number of bytes, words, and lines in files

SYNOPSIS
    wc [-clw] [--bytes] [--chars] [--lines] [--words] [--help]
    [--version] [file...]

DESCRIPTION
    This manual page documents the GNU version of wc.  wc
    counts the number of bytes, whitespace-separated words,
    ...
```

Finding Out What Commands are There

The **apropos** command will search for commands matching a keyword or phrase:

```
(~) 100% apropos column
showtable (1)      - Show data in nicely formatted columns
colrm (1)          - remove columns from a file
column (1)         - columnate lists
fix132x43 (1)     - fix problems with certain (132 column) graphics
modes
```

Arguments and Command Switches

Many commands take arguments. Arguments are often (but not inevitably) the names of one or more files to operate on. Most commands also take command-line "switches" or "options" which fine-tune what the command does. Some commands recognize "short switches" that consist of a single character, while others recognize "long switches" consisting of whole words.

The **wc** (word count) program is an example of a command that recognizes both long and short options. You can pass it the **-c**, **-w** and/or **-l** options to count the characters, words and lines in a text file, respectively. Or you can use the longer but more readable, **--chars**, **--words** or **--lines** options. Both these examples count the number of characters and lines in the text file `/var/log/messages`:

```
(~) 102% wc -c -l /var/log/messages
    23      941 /var/log/messages
```

```
(~) 103% wc --chars --lines /var/log/messages
      23      941 /var/log/messages
```

You can cluster short switches by concatenating them together, as shown in this example:

```
(~) 104% wc -cl /var/log/messages
      23      941 /var/log/messages
```

Many commands will give a brief usage summary when you call them with the **-h** or **--help** switch.

Spaces and Funny Characters

The shell uses whitespace (spaces, tabs and other nonprinting characters) to separate arguments. If you want to embed whitespace in an argument, put single quotes around it. For example:

```
mail -s 'An important message' 'Bob Ghost <bob@ghost.org>'
```

This will send an e-mail to the fictitious person Bob Ghost. The **-s** switch takes an argument, which is the subject line for the e-mail. Because the desired subject contains spaces, it has to have quotes around it. Likewise, my e-mail address, which contains embedded spaces, must also be quoted in this way.

Certain special non-printing characters have *escape codes* associated with them:

Escape Code	Description
<code>\n</code>	new line character
<code>\t</code>	tab character
<code>\r</code>	carriage return character
<code>\a</code>	bell character (ding! ding!)
<code>\nnn</code>	the character whose ASCII code in octal is nnn

Useful Commands

Here are some commands that are used extremely frequently. Use **man** to learn more about them. Some of these commands may be useful for solving the problem set ;-)

Manipulating Directories

ls	Directory listing. Most frequently used as ls -F (decorated listing) and ls -l (long listing).
mv	Rename or move a file or directory.
cp	Copy a file.
rm	Remove (delete) a file.
mkdir	Make a directory
rmdir	Remove a directory
ln	Create a symbolic or hard link.
chmod	

Change the permissions of a file or directory.

Manipulating Files

cat

Concatenate program. Can be used to concatenate multiple files together into a single file, or, much more frequently, to send the contents of a file to the terminal for viewing.

more

Scroll through a file page by page. Very useful when viewing large files. Works even with files that are too big to be opened by a text editor.

less

A version of **more** with more features.

head

View the head (top) of a file. You can control how many lines to view.

tail

View the tail (bottom) of a file. You can control how many lines to view. You can also use **tail** to view a growing file.

wc

Count words, lines and/or characters in one or more files.

tr

Substitute one character for another. Also useful for deleting characters.

sort

Sort the lines in a file alphabetically or numerically.

uniq

Remove duplicated lines in a file.

cut

Remove sections from each line of a file or files.

fold

Wrap each input line to fit in a specified width.

grep

Filter a file for lines matching a specified pattern. Can also be reversed to print out lines that don't match the specified pattern.

gzip (gunzip)

Compress (uncompress) a file.

tar

Archive or unarchive an entire directory into a single file.

emacs

Run the Emacs text editor (good for experts).

Networking

ssh

A secure (encrypted) way to log into machines.

ping

See if a remote host is up.

ftp and the secure version **sftp**

Transfer files using the File Transfer Protocol.

who

See who else is logged in.

lp

Send a file or set of files to a printer.

Standard I/O and Command Redirection

Unix commands communicate via the command line interface. They can print information out to the terminal for you

to see, and accept input from the keyboard (that is, from *you!*)

Every Unix program starts out with three connections to the outside world. These connections are called "streams" because they act like a stream of information (metaphorically speaking):

standard input

This is a communications stream initially attached to the keyboard. When the program reads from standard input, it reads whatever text you type in.

standard output

This stream is initially attached to the command window. Anything the program prints to this channel appears in your terminal window.

standard error

This stream is also initially attached to the command window. It is a separate channel intended for printing error messages.

The word "initially" might lead you to think that standard input, output and error can somehow be detached from their starting places and reattached somewhere else. And you'd be right. You can attach one or more of these three streams to a file, a device, or even to another program. This sounds esoteric, but it is actually very useful.

A Simple Example

The **wc** program counts lines, characters and words in data sent to its standard input. You can use it interactively like this:

```
(~) 62% wc
Mary had a little lamb,
little lamb,
little lamb.

Mary had a little lamb,
whose fleece was white as snow.
^D
      6      20      107
```

In this example, I ran the **wc** program. It waited for me to type in a little poem. When I was done, I typed the END-OF-FILE character, control-D (^D for short). **wc** then printed out three numbers indicating the number of lines, words and characters in the input.

More often, you'll want to count the number of lines in a big file; say a file filled with DNA sequences. You can do this by *redirecting* **wc**'s standard input from a file. This uses the < metacharacter:

```
(~) 63% wc <big_file.fasta
      2943      2998      419272
```

If you wanted to record these counts for posterity, you could redirect standard output as well using the > metacharacter:

```
(~) 64% wc <big_file.fasta >count.txt
```

Now if you **cat** the file *count.txt*, you'll see that the data has been recorded. **cat** works by taking its standard input and copying it to standard output. We redirect standard input from the *count.txt* file, and leave standard output at its default, attached to the terminal:

```
(~) 65% cat <count.txt
```

Redirection Meta-Characters

Here's the complete list of redirection commands for **bash**:

```
<filename      Redirect standard input to file
>filename      Redirect standard output to file
1>filename     Redirect just standard output to file (same as above)
2>filename     Redirect just standard error to file
>filename 2>&1 Redirect both stdout and stderr to file
```

These can be combined. For example, this command redirects standard input from the file named */etc/passwd*, writes its results into the file *search.out*, and writes its error messages (if any) into a file named *search.err*. What does it do? It searches the password file for a user named "root" and returns all lines that refer to that user.

```
(~) 66% grep root </etc/passwd >search.out 2>search.err
```

Filters, Filenames and Standard Input

Many Unix commands act as filters, taking data from a file or standard input, transforming the data, and writing the results to standard output. Most filters are designed so that if they are called with one or more filenames on the command line, they will use those files as input. Otherwise they will act on standard input. For example, these two commands are equivalent:

```
(~) 66% grep 'gatttgc' <big_file.fasta
(~) 67% grep 'gatttgc' big_file.fasta
```

Both commands use the **grep** command to search for the string "gatttgc" in the file *big_file.fasta*. The first one searches standard input, which happens to be redirected from the file. The second command is explicitly given the name of the file on the command line.

Sometimes you want a filter to act on a series of files, one of which happens to be standard input. Many filters let you use "-" on the command line as an alias for standard input. Example:

```
(~) 68% grep 'gatttgc' big_file.fasta bigger_file.fasta -
```

This example searches for "gatttgc" in three places. First it looks in *big_file.fasta*, then in *bigger_file.fasta*, and lastly in standard input (which, since it isn't redirected, will come from the keyboard).

Standard I/O and Pipes

The coolest thing about the Unix shell is its ability to chain commands together into pipelines. Here's an example:

```
(~) 65% grep gatttgc big_file.fasta | wc -l
22
```

There are two commands here. **grep** searches a file or standard input for lines containing a particular string. Lines which contain the string are printed to standard output. **wc -l** is the familiar word count program, which counts words,

lines and characters in a file or standard input. The `-l` command-line option instructs `wc` to print out just the line count. The `|` character, which is known as the "pipe" character, connects the two commands together so that the standard output of `grep` becomes the standard input of `wc`.

What does this pipe do? It prints out the number of lines in which the string "gatttgc" appears in the file *big_file.fasta*.

More Pipe Idioms

Pipes are very powerful. Here are some common command-line idioms.

Count the Number of Times a Pattern does NOT Appear in a File

The example at the top of this section showed you how to count the number of lines in which a particular string pattern appears in a file. What if you want to count the number of lines in which a pattern does **not** appear?

Simple. Reverse the test with the `grep -v` switch:

```
(~) 65% grep -v gatttgc big_file.fasta | wc -l
2921
```

Uniquify Lines in a File

If you have a long list of names in a text file, and you are concerned that there might be some duplicates, this will weed out the duplicates:

```
(~) 66% sort long_file.txt | uniq > unique.out
```

This works by sorting all the lines alphabetically and piping the result to the `uniq` program, which removes duplicate lines that occur together. The output is placed in a file named *unique.out*.

Concatenate Several Lists and Remove Duplicates

If you have several lists that might contain repeated entries among them, you can combine them into a single unique list by **catting** them together, then uniquifying them as before:

```
(~) 67% cat file1 file2 file3 file4 | sort | uniq
```

Count Unique Lines in a File

If you just want to know how many unique lines there are in the file, add a `wc` to the end of the pipe:

```
(~) 68% sort long_file.txt | uniq | wc -l
```

Page Through a Really Long Directory Listing

Pipe the output of `ls` to the `more` program, which shows a page at a time. If you have it, the `less` program is even better:

```
(~) 69% ls -l | more
```

Monitor a Rapidly Growing File for a Pattern

Pipe the output of `tail -f` (which monitors a growing file and prints out the new lines) to `grep`. For example, this will monitor the */var/log/syslog* file for the appearance of e-mails addressed to *mzhang*:

```
(~) 70% tail -f /var/log/syslog | grep mzhang
```
