The "twelve" most important Unix commands

Carl Mason
cmason@berkeley.edu

rev 1.33 Fall 2018

Contents

1 Introduction

Although Unix has a point and click graphic user interface, called X11, which works just like those other operating systems, Unix is at heart a command line operating system. So while it is possible in many cases to do what you want via pointing and clicking, using the command line and other text based tools will make you happier and **much** more efficient... eventually.

To operate with the command line, you will need to know the 12 most important Unix commands described in Section 5. To enjoy it you will also need to know a few tricks that are also covered in this document.

You don't **need** to know much about Unix in order to start doing Science, but it would not hurt to learn more. In your copious free time, check out

some of the Unix primers on the web. Ask google something like "Unix beginner" to find more resources than you could possibly want.

Note that since the Mac OS is simply a Unix application, nearly everything in this document works the same way in a mac. On a mac, the terminal window application is under `Applications/Utilities`.

2 Terminal windows

In order to use the command line or ***shell***, you must open a ***terminal window*** (also known an ***xterm*** window). There are several very similar terminal window applications which for our purpose are completely interchangeable. ***roxterm*** and ***mate-terminal*** are two that you will find under `[Application]`→`[System Tools]`.

A terminal window should start out looking something like Figure 1. Notice that the window features a menu bar – as you'll discover, by reading the rest of this sentence, the menu bar is only useful when you want to fiddle with the terminal windows many many configuration options. This is something you will only want to do when you are actively trying to avoid doing something useful – so your best option is to use the `RIGHT` to reveal a menu that will allow you to NOT "Show Menu Bar". When it's time to study for prelims, you can expose the menu bar again and fiddle with fonts and background colors and chirps and beeps and whatever.

Aside from the title bar at the top, the only words in the terminal window should be the ***Unix prompt***. The purpose of the Unix prompt is to indicate that the ***shell*** is ready to accept commands. It also contains useful information. In Figure 1, the prompt is is `[carlm@twins ~]$`, indicating the user, `carlm`, the machine, `twins` and the current directory which is indicated by the `~`. In this and other documents, the Unix prompt will look like this: **@:>** . In the real Unix prompt, the symbol `~` is a special character whose meaning is "home directory". `~/Dissertation` means a file or directory called "Dissertation" which is located within your home directory. In my case this would be `/hdir/0/carlm/Dissertation`. `~wachter/Brilliant/insight` translates to a file (or possibly a directory) called `insight` in a directory[1] called `Brilliant` in Ken Wachter's home directory, or `/hdir/0/wachter/Brilliant/insight`. More about home directories can be found in Section 2.

Although you are too young now for this to matter, someday, if you are lucky the default font size in the terminal window and elsewhere will become

[1] directories are also called "folders"

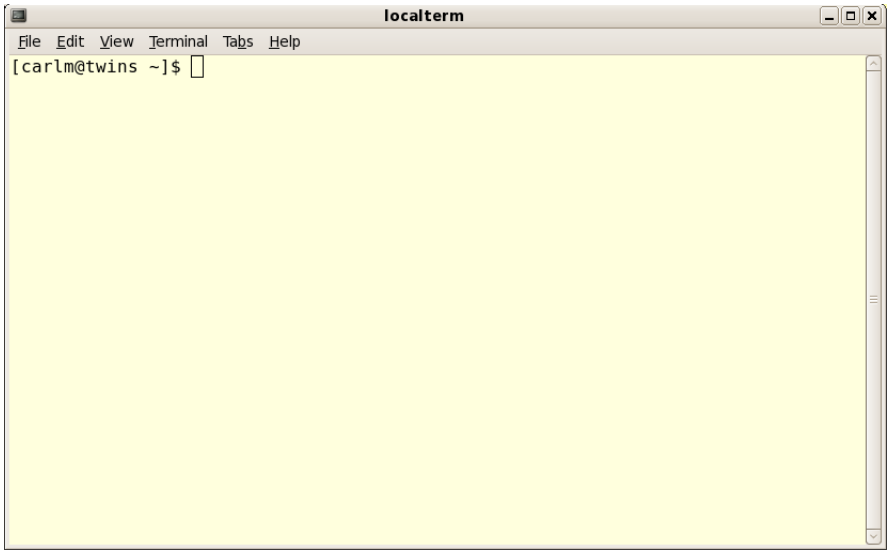

Figure 1: terminal window

too small to read –even through the bottom of you thick progressive lenses. To prolong your career at that point, a useful trick with is the CTRL +SHIFT ++ to increase and CTR +- to decrease the size of the typeface. This also works in browsers and many other applications. Macs and windows machines have something similar.

3 The Filesystem

Whenever you login to a machine on the Demography network, your initial ***present working directory*** – the location within the filesystem in which applications will begin looking for the files that you specify – is your home directory. Every user has exactly one home directory.

In a multiuser system such as the Demography Lab, your home directory is one of a huge number of interconnected directories that form a single unified ***filesystem***. The magic of the filesystem is that even though the various files and directories of which it is composed are "physically"/footnoteor electromagnetically present on various different machines all over the network, to us users, the whole thing appears to be one single thing and that thing looks and feels the same no matter which Demography Lab machine we happen to be using at the moment.

An upside down tree makes a pretty good metaphor for the filesystem.

Such a "tree" is shown in Figure 2. At the top of the figure is a directory called "/" which is the "root" of the filesystem. Every file and directory in the filesystem can be uniquely specified by a ***filepath*** that begins with root. For example, the file that holds my correspondence with my mother might be `/hdir/0/carlm/mail/mom`.

As you can see in Figure 2 home directories all live in a directory called `/hdir/0`. Although it is just one of many directories within this giant upside down tree of a filesystem, your home directory is a special place that you will come to know and love and where you will do your very best work. It is the part of the filesystem that you own and the "place" where you will find yourself when you first login.

Because the entire filesystem looks the same to all users all the time, it is easy to share data with your colleagues. This is good thing because humanity benefits when scientists collaborate. But unfortunately scientists can occasionally turn out to be creeps so sharing a filesystem is a little scary as well.

The "solution" to the creep problem is to not keep sensitive information on Demography computers. You have already promised not to keep data covered by SB 1386[2]. It goes without saying that files that can tie you to illegal activities are also a no-no. There are however, a few files that belong on the network and yet where privacy is an issue. For those files, managing who may read and/or change them requires understanding the mode and ownership of files. Each file and directory has an owner and the owner can determine who is allowed to read, write and/or execute each file. See the `chmod` command below for how to change the various file ***modes*** or ***permissions***. The `chmod` command is described in 6.

4 The command interpreter (or shell)

The command interpreter, or *shell* is the program that runs in each terminal window. It waits for you to type something at the ***Unix prompt***, **@:>** , and then does what *it* thinks you meant. The shell we use here is called *bash* (pronounced "bash"). Bash is one of several modified versions of the original sh (pronounced "s-h") shell.

The most important thing that the shell does for you is to let you give commands to the computer. These include the 12 most important Unix commands (Section 5) as well as commands to launch applications like R,

[2]See the statement of compliance that you signed before we gave you an account

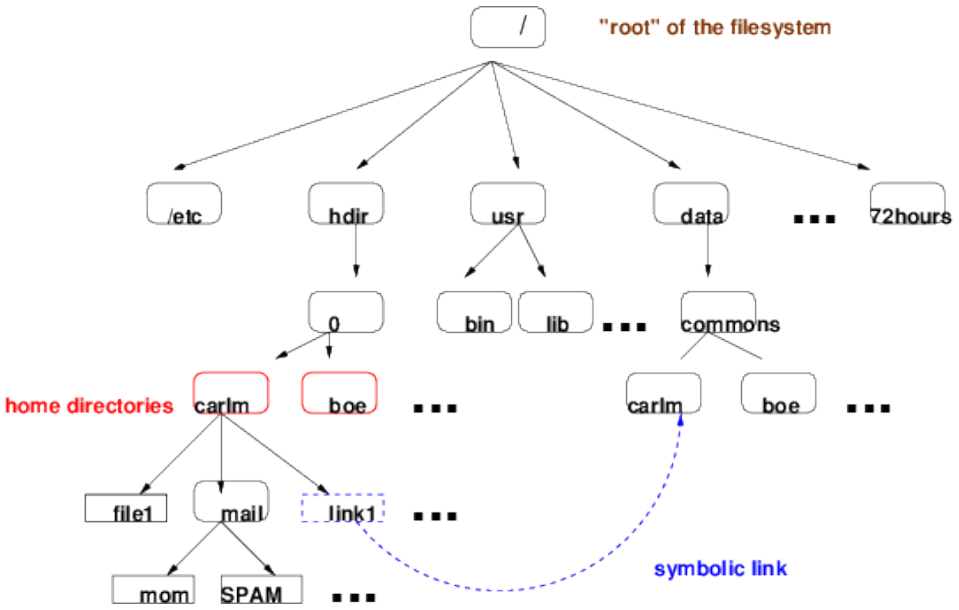

Figure 2: The Demography Lab filesystem

Stata, word processors or spreadsheets[3]. The shell does several other things for you some are essential, some enhance efficiency and others are just cool.

4.1 Essential stuff

Among the essential features of the shell is a mechanism to communicate with running programs that are not expecting user input or have run amok. This is not all that common, but when it happens you need to be able to get the program's attention and tell it – generally to drop dead.

4.1.1 Killing stuff

To kill a program that happens to be running in the foreground of a terminal window – e.g. you launched it from the command line and the terminal window that you launched from is not showing a prompt – you can simply hit CTRL + C . This mostly works, as does closing the terminal window.

[3]It is of course possible to launch most applications via the menu system or by clicking on corresponding data files in the file manager, but the command line is often faster

A more general approach is useful however because often applications are started via menus or are for other reasons running in the background (independent of a terminal window). To kill such a program requires knowing its ***process id*** or PID.

Your friend for finding a PID is ***ps***

@:> `ps -ux`

produces a list of all of your running processes :

```
carlm@immigrant:~$ ps -ux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
carlm      568  0.0  0.0  35280  3112 pts/9    R+   12:49   0:00 ps -ux
carlm    27216  0.0  0.0  43192  5156 ?        Ss   08:39   0:00 /lib/systemd/systemd --user
carlm    27221  0.0  0.0  77800  2288 ?        S    08:39   0:00 (sd-pam)
carlm    27262  0.0  0.0 105468  5828 ?        R    08:39   0:07 sshd: carlm@pts/9
carlm    27263  0.0  0.0   4504   752 pts/9    Ss   08:39   0:00 /bin/sh -c /bin/bash
carlm    27269  0.0  0.0  21448  6328 pts/9    S    08:40   0:00 /bin/bash
carlm    27528  0.0  0.0  42608  2308 pts/9    S    08:42   0:00 dbus-launch --autolaunch a8f1d898ba4dba1bb
carlm    27529  0.0  0.0  40824  3244 ?        Ss   08:42   0:00 /usr/bin/dbus-daemon --fork --print-pid 5
carlm    27531  0.0  0.0  59216  5488 ?        S    08:42   0:00 /usr/lib/x86_64-linux-gnu/gconf/gconfd-2
carlm    32356  0.7  0.1 571456 67064 pts/9    Sl   12:25   0:10 emacs 12important.tex
carlm    32511  0.0  0.0   4504   704 ?        Ss   12:27   0:00 /bin/sh -c okular --unique --page 2 12impo
carlm    32512  0.1  0.1 605712 91464 ?        Sl   12:27   0:01 okular --unique --page 2 12important.pdf
carlm    32521  0.0  0.0 178056 16172 ?        Ss   12:27   0:00 kdeinit4: kdeinit4 Running...
carlm    32524  0.0  0.0 200472 17968 ?        S    12:27   0:00 kdeinit4: klauncher [kdeinit] --fd=8
carlm    32527  0.0  0.0 272180 27192 ?        S    12:27   0:00 kdeinit4: kded4 [kdeinit]
carlm    32553  0.0  0.0 542152 24316 ?        Sl   12:27   0:00 /usr/bin/kactivitymanagerd start-daemon
carlm@immigrant:~$
```

Many of the columns above are unlikely to interest you, but there two that will: The "COMMAND" column shows the command that launched the process "emacs 12important.tex" in the 10^{th} row is the entry associated with the emacs editor program which I am using to edit this file right now.

If I wanted to kill that program and thereby lose many minutes of editing effort, I could type:

@:> `kill 32356`

in any shell window. 32356 comes from the column called "PID". The `kill` command with no argument other than a PID is a polite sort of kill – much like pulling down the "file" menu and selecting "exit" in many applications. When that does not work, `kill -9 PID` is much more aggressive. It causes program to stop without saving or cleaning up in general.

4.2 Efficient stuff

To make you more efficient, the shell offers three particularly nice features: "history", "TAB completion" and "scripting".

history The history feature allows you to recall **and edit** any command that you have previously issued. To make the **p**revious command appear at the **@:>** hit CTRL +P or equivalently **the up arrow key**. To see even more previous stuff type CTRL +P more times. CTRL +N or equivalently **the down arrow** will make the **n**ext command appear – obviously, this makes no sense unless you have typed CTRL +P at least once.

You can operate on a recalled command using several standard **emacs** editing keys:

- CTRL +A To go to the beginning of the current line
- CTRL +E To go to the end of the current line
- CTRL +F To go forward one character
- CTRL +B To go backward one character

You can also use the **left arrow** and **right arrow** to move about within a recalled line. The DELETE and BACKSPACE keys do what you would expect.

TAB completion If you hit the TAB key anytime while constructing a command, the shell will do it's best to figure out what you *are planning to type next*. If you are typing a command it will try to find a command that starts out with what you have already typed. If you are typing the name of a file the shell will try to complete if for you. If what you have typed does not uniquely determine a command or filename, the shell will beep at you and provide a list of possible completions. You can then type a few more characters and hit TAB again.

scripting Whenever you find yourself typing the same command several times, it's time to consider scripting. A shell script is just a file of commands that you *could have* entered at the keyboard, but typed into a file instead. You can then set the file's execute bit (See Most Important Command number 6) and execute that file – perhaps now, perhaps later. You will need to use an editor such as emacs to create that shell script. Knowing how to use emacs can save you lots of time and hair loss – particularly if many of the commands you are typing are quite similar.

Scripts are also very useful for people who like the idea of being able to reproduce results.

cool stuff The shell is also responsible for displaying the results of the `ls` command (See 1) in lots of colors.

glob expressions are combinations of letters and special characters that the shell interprets in clever ways (Glob is a very simplified version of Regular Expressions the concept is so similar that I will accidentally use the term Regular Expression when I mean glob expression). In the shell we use glob expressions to specify lists of files or directories on which a command should operate. A typical use would be to delete from your current working directory all of the .pdf files whose name begins with a vowel:

```
@:> rm [AEIOUaeiou]*.pdf
```

The letters between the [square brackets] form a list of which any included element will be considered a "match" because nothing proceeds the [] in order for a file to match, it must begin with one of the letters in the square brackets. the '*' means zero of more of any character. Glob expressions come up in several of the "12" important commands.

5 The 12 most important Unix commands

Below is a list of the 12 most important Unix commands. They are simple enough to be easily memorized by anyone who can keep the names of all twelve months in his head.

For the most part, these commands are for logging on and off; for printing; and for moving files and directories around. Many of these commands functions can be done using a file manager or under emacs, but knowing how to do them from the command line, makes you more efficient, reduces errors and opens the possibility of automating tasks with *shell scripts*.

> NOTE the <`angle brackets`> indicate that a command argument is *optional* you do not type the <>'s it's just a typographical convention

1. `ls` <`-ltr`> <`glob expression`> The "list file" command, ls is used –not surprisingly– to list the names and pertinent information about some or all of the files in a particular directory. The most common and useful option is `-l` that's a lower case L not a one. It reveals the most interesting properties of your files. Adding ***tr*** causes ls to present it results sorted by **t**ime in **r**everse order.

```
@:> ls -ltr directory1
```

produces a list of files sorted so that the most recent ones are at the bottom of the list.

See 4.2 for a description of glob expressions.

2. `mkdir <-p> new-directory-name` The "make directory" command is used to create a new sub-directory of the current working directory. The ***-p*** argument causes `mkdir` to create "parent" directories as needed. In other words,

```
@:> mkdir -p first/second/third
```

would create a a directory called `third` which would be a subdirectory of `second` which in turn would be a subdirectory of `first`. The `-p` argument instructs mkdir to create `second` and/or `first` if they do not already exist.

3. `cd <directory-name>` The "change directory" command makes another directory your present working directory. With no argument, it "moves you" to your home directory. To move one directory "higher" use ".." (two dots) in place of the directory's name. The one and only parent directory of the current directory is always addressable as "..".

4. `cp <-R>` source target

   The copy command, `cp` is used to copy files or a directory full of files.

   - To copy a file you specify the name of an existing file as the "source" and you specify a legal filename as the "target".
   - If you want the new copy of the file to be in a different directory, then you can specify a path_to_an_existing directory as the "target".
   - To copy a directory full of files you use the -R argument. As is often the case, the "-R" stands for "recursive".

   ```
   @:> cp -R directory1 directory2
   ```

   The above command will copy directory1 and all the files and subdirectories contained therein into directory2. If directory2 doesn't exist, it will be created, if it does exist, then this command will create a subdirectory of `directory2` with the same name as `directory1` and containing copies of all of the files and directories in `directory1`.

5. `rm file-name` **or** <`regular expression`> The ***rm*** command is used to remove, or erase files. Here again, regular expressions can be very useful – and quite dangerous.

   `rm` is often ***aliased***[4] to `rm -i`, so that it asks you to verify that you really want to remove a file. If you get tired of this safety feature, use `\rm` instead.

   Note: `rm` will accept a `regular expression` as an argument. The simplest regular expression is "*" which stands for everything in the present working directory. So be careful.

   To remove an entire directory and and all the files and subdirectories in it, you use the ***-r*** argument and the leading back slash, "\":

   @:> -rf directory1

   the above command will remove directory1 and all of the files and subdirectories within it, the ***-f*** argument ensures that rm will not ask for permission with each file. -rf * is a **VERY** dangerous command. If you find yourself typing it make sure you are not drunk.

   > Heads up: the rm command really and truly removes a file. **rm cannot be undone**. This is different from the way the file manager moves stuff to the "trash".

6. `chmod` <`aogu +/- rwx`> filename-or-directory The "change mode" command is used to modify the permissions (or mode) of a file or directory. Permissions are the characteristic of a file or directory which determine who has what type of access to it. All files and all directories have permissions, only the owner of the file/ directory is permitted to change modes.

   The first argument is a string of characters that grant (+) or revoke (-) permission to read(r) write(w) or execute (x) the file or directory. The letters aogu indicate who is to receive or lose the given permission. u=user, g=group, o=other, and a=all. Thus to revoke write permission to all users you would type:

[4] as you might expect, "alias" is a shell feature that allows you to create new names for commands. It is possible and common to use this feature so that when you type `rm` the shell substitutes `rm -i`. The `-i` argument causes rm to ask for verification before it removes a file

`chmod a-w filename`

To grant permission to yourself and the group to write and execute a file you would type:

`chmod ug+wx filename`

7. `ln <-s> real-file-name artificial-file-name` The "link" command creates an alternative name for an existing file or directory. This is particularly useful when using data sets that you keep in /data/commons (as you should). Rather than typing /data/commons/userid/datafile to reference your data, a symlink would allow you to type something much shorter.

   `ln -s /data/commons/userid DATA`

   would create a link in your current directory called "DATA". But DATA is really just a secret back way to /data/commons/userid. typing `ls DATA` for example is the equivalent of typing `ls /data/commons/userid`.

   It would be a good idea to create the above link right now. Use the `mkdir` to create a new directory in /data/commons called your userid. Then create a link in your home directory so that you can start storing and accessing huge data sets right away.

8. `mv file-name new-file-name` The "move" command changes the name or location within the filesystem of a file or a directory.

9. `less file-name` Variant of the `more` command – less is used to scroll through a file on the screen. While displaying a file, ENTER scrolls one additional line; SPACE scrolls one additional screen full; B scrolls backwards, Q quits, /WORD searches forward for "word", ?WORD searches backward for "word".

10. `gtklp filename` launches a gui application which allows you to print `filename` to any Demography of Sociology printer command prints a file to the named printer. Most of the time we print from with applications so this command is not so frequently used anymore. Also since it launches a gui application, it is not strictly speaking a Linux command. For purists and dinosaurs the Linux command in `lpr`.

Demography and Sociology Department Printers

Printer	Location	Type
age	Basement Lab	HP Laserjet 4015 postscript monochrom duplex 1200dpi
region	2224 2nd floor	HP 4100n postscript 600dpi
cohort	2232 2nd floor	HP 4200n postscript 600dpi
reproduction	2232 1st floor	Canon Image Runner Advanced C5235
Barrows477	Rm 477 Barrows	Xerox phaser
Barrows483	Rm 483 Barrows	Xerox phaser

11. `pwd` "present working directory" tells you where you are, that is, it tells you which directory the shell thinks is the current directory.

12. `du <directory>` The "disk use" command is designed to tell you how much disk space each directory is consuming. It's main use, however, is simply to display the directory structure.

12. `man -k key-word` — `command-name` The "manual" command is used to display manual pages on your screen. To say that man pages are not particularly easy to read is is an understatement of almost biblical magnitude. But they are very handy for refreshing your memory or searching for something very specialized.

    The man program puts the contents of the man page in a "less" process, see item 9 for a description of how to navigate in less.

    In this century a very good source of information on Unix is google. The web knows all about Unix and while there are lots of different distributions, command line tools in particular are nearly identical in all distributions including Solaris, Hpux, every flavor Linux, BSD, and even Apple's OS X (which by the way is Unix too).

12. `ssh <-l userid> hostname` "ssh" stands for "secure shell" it is really a separate application but it behaves like a shell command and is really useful so it is included here. If you type `ssh keyfitz` at the unix prompt, (and then your password when prompted) a remote shell will open on an entirely different machine from the one you are sitting in front of.

    The new remote shell on keyfitz will have a prompt like: `[userid@keyfitz ~]$` indicating that the commands that you type will be executed the machine `keyfitz` which happens to live in the server room in 2224 Piedmont. Happily the new shell will see the same filesystem and understand the same Unix commands.

The reason for ssh'ing to keyfitz is that it is much more powerful than immigrant (which power the workstations) or quigley which powers noMachine. Also, keyfitz does not run any desktops so other users will be minimally inconvenienced by your humongous multi hour job.

NOTE: when using ssh or ssh-like programs on machines outside of the Demography Lab, you will need to specify both your Demography Lab userid as in `ssh carlm@nmx.demog.berkeley.edu`[5]. Also – ssh works from a mac but with windows you need some other application.

NOTE even more urgently: Since we started running noMachine ssh'ing from outside of the department instantly became anachronistic. noMachine provides a generally better way of connecting to the Demography network if your goal is to do science. `http://lab.demog.berkeley.edu/LabWiki` is the place to go to find out about noMachine and much else.

12. `exit` or `logout` closes the current Unix window, and logs you off – if the current window is the console window.

6 Special and "meta" characters

In addition to the key combinations and commands discussed, Unix also supports several characters with special meanings to the shell. Below is a list some of the more common ones:

***** The asterisk or "star" character is used in glob expressions (See item 4.2). When the shell sees a * by itself as in **@:>** `ls *` it replaces * with a list of all the files and subdirectories in the current directory. **@:>** `ls *` tells the shell to run the ls command on each and every file and subdirectory in the current directory. So where **@:>** `ls` will show files and subdirectories **@:>** `ls *` will list the files that live *in subdirectories* of the current directory as well.

& The ampersand tells the shell to run the process in the "background". When a process is launched in the background, the xterm (See 2) immediately returns with a prompt. When you run a process in the foreground (the usual case) the prompt comes back only when the process exits.

[5] surprise - from outside the department you will probably end up on refugee rather than quigley if you specify the host as @demog.berkeley.edu

NOTE it only makes sense to run programs in the background if the program spawns a new window. So `emacs`, `Stata`, userfirefox, or `libreoffice` are all fine running in the background. The 12 most important Unix commands are not. They all write their responses to the terminal window. If you put them in the background they cannot do this.

REALLY important: R should not be run with the & for the same reason: it runs in the window from which it was launched. This will all make sense after the first week or two of 213.

To bring a backgrounded program to the foreground, type

@:> `fg <%n>`

where %n is the percent sign followed by a number indicating which backgrounded process you want to foreground. You only need to enter the %n if you have more than one process running in the background. Type **@:>** `jobs`

to get a list of backgrounded processes associated with the current xterm.

. (dot) The dot is interpreted by the shell to mean *the current working directory*. So expressions like like `./insight.py` the file `insight.py` in the current directory.It's generally not necessary if your are passing a filepath to a Linux command for example

@:> `cat insight.py`

and

@:> `cat ./insight.py`

will both print the contents of the same file to the screen.

Where the dot is important is when insght.py is executable – perhaps a python program that you just finished writing. For security reasons, the shell will only look for executable files in a prespecified set of locations in the filesystem. Even though insight.py is sitting right there in your current directory, the if you ask the shell to execute it, it will not be found. Unless you refer to it as `./insight.py`

Watch the flyspecks: In addition to representing the current working directory, "." also has the effect of making things invisible. File and directory names that begin with "." are traditionally configuration

files – stuff that you don't generally want showing up everytime you `ls`. Consequenlty, `.insight.py` refers to something entirely different from `./insight.py`.

..(two dots) are interpreted as the parent of the current directory.

~ the tilde character is interpreted by the shell to mean "home directory" by itself, it means **your** home directory, if it is followed by a username as in ~carlm it refers to that user's home directory. The ~ can be used in complicated pathnames such as ~`carlm/public_html/213F97/welcome.html`. For it to make sense, the ~ must be the first character (and perhaps the only character) of a pathname.

| The "pipe" is used to send the standard output of one process into the standard input of another. For example, if you wanted to know the number of lines in every data file in the current directory you might type: **@:>** `ls *.data | wc -l` . The `ls *.data` produces a list of files in the current directory that end in ".data", the | then feeds this list to the **w**ord **c**ount command "wc". The `-l` argument tells `wc` to only report the number of lines. This example assumes that you have named all of your data files *somethingorother*.data.

> The right angle bracket (or greater than sign) is used to send the output of a process into a file. **@:>** `ls > file.list` would produce a file called file.list containing (surprise) a list of files. Use double angle brackets to append a process's output to an existing file.

@:> `ls ~/public_html >> file.list`

would add the names of the file's in your public_html directory.