

The Command Line

Whatever operating system you are on, I assume you have access to some Unix-like command line. That is, you are looking at a blank screen (probably) with a prompt like `$` waiting for you to type something. You could try typing “hello” and then the Enter key, and you’ll probably see something like this:

```
$ hello
-bash: hello: command not found
```

NB: When you see a `$` given in the example prompts, it is a metacharacter indicating that this is the prompt for a normal (not super-user) account. You should type/copy/paste all the stuff *after* the `$`. If you ever see a prompt with “`#`” in a tutorial, it’s indicating a command that should be run as the super-user/root account, e.g., installing some software into a system-wide directory so it can be shared by all users.

The first word on the command line needs to be a *command*. On my system, there’s no program or built-in command called `hello` anywhere to be found (specifically in the `$PATH`, but we’ll get to that in a bit), so it tells me **command not found**. Try `help` instead and you’ll likely see quite a bit of output if you are on a system that is running GNU’s `bash` shell. The command `hostname` should work pretty evrerywhere, telling you the name of your machine. Certainly `ls` will work to show you a directory listing. Try it!

Common Unix Commands

“The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do.” - Ted Nelson

Let’s look at some more commands you can do. This is by no means an exhaustive list, just a few to get you going. Try running each of them. To learn more about the tools, try both `man cmd` or `cmd -h` or `cmd --help`.

- **whoami**: reports your username
- **w**: shows who is currently on a system
- **man**: show the manual page for a command
- **echo**: say something
- **cowsay**: have a cow say something
- **env**: print your environment
- **printenv**: print some/all of your environment
- **which/type**: tells you the location of a program
- **touch**: create an empty regular file
- **file**: briefly describe a file or directory
- **pwd**: print working directory, where you are right now

- **ls**: list files in current directory
- **find**: find files or directories matching some conditions
- **locate**: find files using a database, requires daemon to run
- **cd**: change directory (with no arguments, `cd $HOME`)
- **cp**: copy a file (or “`cp -r`” to copy a directory)
- **mv**: move a file or directory
- **mkdir**: create a directory
- **rmdir**: remove a directory
- **rm**: remove a file (or “`rm -r`” to remove a directory)
- **cat**: concatenate files (cf. <http://porkmail.org/era/unix/award.html>)
- **column**: arrange text into columns
- **paste**: merge lines of files
- **sort**: sort text or numbers
- **uniq**: remove duplicates *from a sorted list*
- **sed**: stream editor for altering text
- **awk/gawk**: pattern scanning and processing language
- **grep**: global regular expression program (maybe?), cf. https://en.wikipedia.org/wiki/Regular_expression
- **history**: look at past commands, cf. CTRL-R for searching your history directly from the command line
- **head**: view the first few (10) lines of a file
- **tail**: view the last (10) lines of a file
- **comm**: find lines in common/unique in two sorted files
- **top**: view the programs taking the most system resources (memory, I/O, time, CPUs, etc.), cf. “htop”
- **ps**: view the currently running processes
- **cut**: select columns from the output of a program
- **wc**: (character) word (and line) count
- **more/less**: pager programs that show you a “page” of text at a time; cf. <https://unix.stackexchange.com/questions/81129/what-are-the-differences-between-most-more-and-less/81131>
- **bc**: calculator
- **df**: report file system disk space usage; useful to find a place to land your data
- **du**: report disk usage; recommend “`du -shc`”; useful to identify large directories that need to be removed
- **ssh**: secure shell, like telnet only with encryption
- **scp**: secure copy a file from/to remote systems using ssh
- **rsync**: remote sync; uses scp but only copies data that is different
- **ftp**: use “file transfer protocol” to retrieve large data sets (better than HTTP/browsers, esp for getting data onto remote systems)
- **ncftp**: more modern FTP client that automatically handles anonymous logins
- **wget**: web get a file from an HTTP location, cf. “wget is not a crime” and Aaron Schwartz
- **|**: pipe the output of a command into another command

- **>, >>:** redirect the output of a command into a file; the file will be created if it does not exist; the single arrow indicates that you wish to overwrite any existing file, while the double-arrows say to append to an existing file
- **<:** redirect contents of a file into a command
- **nano:** a very simple text editor; until you're ready to commit to vim or emacs, start here
- **md5sum:** calculate the MD5 checksum of a file
- **diff:** find the differences between two files
- **xargs:** take a list from one command, concatenate and pass as the arguments to another command

The Unix filesystem hierarchy

The Unix filesystem can be thought of as a graph or a tree. The root is “/” and is called the “root directory.” We can “list” the contents of a directory with **ls**. Without any arguments, this prints the contents of the current working directory which you can print with **pwd** (print working directory). You can **ls /** to see the contents of the root directory, or **ls \$HOME** to see your own home directory.

Moving around the filesystem

You can print your current working directory either with **pwd** or **echo \$PWD**.

The **cd** command is used to “change directory,” e.g., **cd /rsgrps/bh_class/**. If you wish to return to the previous working directory, use **cd -**.

If you provide no argument to **cd**, it will change to your **\$HOME** directory which is also known in bash by the **~** (tilde or twiddle). So these three commands are equivalent:

- **cd**
- **cd ~**
- **cd \$HOME**

Once you are in a directory, use **ls** to inspect the contents. If you do not provide an argument to **ls**, it assumes the current directory which has the alias **.**. The parent directory is **..**.

You can use both absolute and relative paths with **cd**. An absolute path starts from the root directory, e.g., “/usr/local/bin/”. A relative path does not start with the leading / and assumes a path relative to your current working directory. If you were in the “/usr/local” directory and wanted to change to “/usr/local/bin”, you could either **cd /usr/local/bin** (absolute) or **cd bin** (relative to “/usr/local”).

Once you are in “/usr/local/bin”, what would `pwd` show after you did `cd ../../?`

Chaining commands

“Programming is breaking of one big impossible task into several very small possible tasks.” - Jazzwant

Most Unix commands use STDIN (“standard in”), STDOUT (“standard out”), and STDERR (“standard error”). For instance, the `env` program will show you key/value pairs that describe your environment – things like your user name (`$USER`), your shell (`$SHELL`), your current working directory (`$PWD`). It can be quite a long list, so you could send the output (STDOUT) of `env` to `head` to see just the first few lines:

```
$ env | head
TERM_PROGRAM=iTerm.app
TERM=xterm-256color
SHELL=/bin/bash
TMPDIR=/var/folders/0h/vjzky052qx4p70trn2p2h400000gn/T/
PERL5LIB=/Users/kyclark/work/imicrobe/lib
Apple_PubSub_Socket_Render=/private/tmp/com.apple.launchd.MoI0Cra0uS/Render
TERM_PROGRAM_VERSION=3.2.6
OLDPWD=/Users/kyclark/work/biosys-analytics/lectures
TERM_SESSION_ID=w2t0p0:34E89330-9CEE-4FA6-9039-3674CBDE4655
USER=kyclark
```

If you provide a file name as an argument to `head`, it will work on that:

```
$ head /usr/share/dict/words
A
a
aa
aal
aalii
aam
Aani
aardvark
aardwolf
Aaron
```

Without any arguments, `head` assumes you must want it to read from STDIN. Many other programs will assume STDIN if not provided an argument. For instance, you could pipe `env` into `grep` to look for lines with the word “TERM” in them:

```
$ env | grep TERM
TERM_PROGRAM=iTerm.app
```

```

TERM=xterm-256color
TERM_PROGRAM_VERSION=3.2.6
TERM_SESSION_ID=w2t0p0:34E89330-9CEE-4FA6-9039-3674CBDE4655
ITERM_PROFILE=Default
ITERM_SESSION_ID=w2t0p0:34E89330-9CEE-4FA6-9039-3674CBDE4655
COLORTERM=truecolor

```

If you are fortunate enough to have the `fortune` and `cowsay` programs on your system, you can do this:

```

$ fortune | cowsay
-----
/ Somebody ought to cross ball point pens \
| with coat hangers so that the pens will |
\ multiply instead of disappear.           /
-----
      \   ^__^
       \  (oo)\_______
          (__)\       )\/\
             ||----w |
             ||     ||

```

Manual pages

“Programming isn’t about what you know; it’s about what you can figure out.” - Chris Pine

The `man` program will show you the manual page for a program, if it exists. Just type `man <program>`, e.g., `man wget`. Inside a manpage, you can use the `/` to search for a string. Use `q` to “quit” `man`. Most programs will also show you a help/usage document if you run them with `-h`, `--help`, or `-help`. I often find it useful to `grep` the help, e.g.:

```

$ wget --help | grep clobber
  -nc, --no-clobber          skip downloads that would download to
  --unlink                  remove file before clobber

```

Pronunciation

- `/`: “slash”; the thing leaning the other way is a “backslash”
- `sh`: “shuh” or “ess-ach”
- `etc`: “et-see”
- `usr`: “user”
- `src`: “source”

- #: “hash” (NOT “hashtag”) or “pound”
- \$: “dollar”
- !: “bang”
- #!: “shebang”
- ^: “caret”
- **PID**: “pid” (not pee-eye-dee)
- ~: “twiddle” or “tilde”; shortcut to your home directory when alone, shortcut to another user’s home directory when used like “~bhurwitz”

Variables

You will see things like `$USER` and `$HOME` that start with the `$` sign. These are variables because they can change from person to person, system to system. On most systems, my username is “kyclark” but I might be “kclark” or “kyclark1” on others, but on all systems `$USER` refers to whatever value is defined for my username. Similarly, my `$HOME` directory might be “/Users/kyclark,” “/home1/03137/kyclark,” or “/home/u20/kyclark,” but I can always refer to the idea of my home directory with the variable `$HOME`.

When you are assigning a variable, you do not use the `$`.

```
[hpc:login3@~]$ SECRET=ilikecake
[hpc:login3@~]$ echo $SECRET
ilikecake
[hpc:login3@~]$ echo SECRET
SECRET
```

To remove a variable from your environment, use `unset`:

```
[hpc:login3@~]$ unset SECRET
[hpc:login3@~]$ echo $SECRET
```

Notice that there is no error when referencing a variable that does not exist or has not been set.

Control sequences

If you launch a program that won’t stop, you can use CTRL-C (where “CTRL” is the “control” key sometime written “`^C`” or “`^-C`”) to send an “interrupt” signal to the program. If it is well-behaved, it should stop, but it may not. For example, perhaps I’ve tried to use a text editor to open a 10G FASTA file and now my terminal is unresponsive because the editor is using all available memory. I could open another terminal on the machine and run `ps -fu $USER` to find all the programs I am running:

```
$ ps -fu $USER
UID      PID  PPID  C  STIME TTY          TIME CMD
kyclark  31718 31692  0  12:16 ?           00:00:00 sshd: kyclark@pts/75
kyclark  31723 31718  0  12:16 pts/75     00:00:00 -bash
kyclark  33265 33247  0  12:16 ?           00:00:00 sshd: kyclark@pts/86
kyclark  33277 33265  1  12:16 pts/86     00:00:00 -bash
kyclark  33792 33277  9  12:17 pts/86     00:00:00 vim maize_genome.fasta
kyclark  33806 31723  0  12:17 pts/75     00:00:00 ps -fu kyclark
```

The PID is the “process ID” and the PPID is the “parent process ID.” In the above table, let’s assume I want to kill vim, so I type kill 33792. If in a reasonable amount of time (a minute or so) that doesn’t work, I could use kill -9 (but it’s considered a bit uncouth).

CTRL-Z is used to put a process into the background. This could be handy if, say, you were in an editor, you make a change to your code, you CTRL-Z to background the editor, you run the script to see if it worked, then you fg to bring it back to the foreground or bg it to have it resume running in the background. I would consider this a sub-optimal work environment, but it’s fine if you were for some reason limited to a single terminal window.

Generally if you want a program to run in the background, you would launch it from the command line with an ampersand (“&”) at the end:

```
$ my-background-prog.sh &
```

Lastly, most Unix programs interpret CTRL-D as the end-of-input signal. You can use this to send the “exit” command to most any interactive program, even your shell. Here’s a way to enter some text into a file directly from the command line without using a text editor. After typing the last line (i.e., type “chickens.<Enter>”), type CTRL-D:

```
$ cat > wheelbarrow
so much depends
upon

a red wheel
barrow

glazed with rain
water

beside the white
chickens.
<CTRL-D>
$ cat wheelbarrow
so much depends
upon
```

```
a red wheel  
barrow  
  
glazed with rain  
water  
  
beside the white  
chickens.
```

Handy command line shortcuts

- Tab: hit the Tab key for command completion; hit it early and often!
- **!!**: (bang-bang) execute the last command again
- **!\$**: (bang-dollar) the last argument from your previous command line (think of the \$ as the right anchor in a regex)
- **!^**: (bang-caret) the first argument from your previous command line (think of the ^ as the left anchor in a regex)
- CTRL-R: reverse search of your history
- Up/down cursor keys: go backwards/forwards in your history
- CTRL-A, CTRL-E: jump to the start, end of the command line when in emacs mode (default)

NB: If you are on a Mac, it's easy to remap your (useless) CAPSLOCK key to CTRL. Much less strain on your hand as you will find you need CTRL quite a bit, even more so if you choose emacs for your \$EDITOR.

Altering your \$PATH

I feel like there should seriously be some sort of first-year level class where people learn how to add something to their path and what that means. – Kristopher Micinski

Your \$PATH setting is an ordered, colon-delimited list of directories that will be searched to find programs. Run `echo $PATH` to see yours. It probably looks something like this:

```
$ echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

Here's my \$PATH on my laptop:

```
$ echo $PATH | gsed "s:/\n/g"  
/Users/kyclark/.cargo/bin  
/Users/kyclark/bin  
/anaconda3/bin
```



```
/Users/kyclark/.local/bin
/Users/kyclark/work/cyverse-cli/bin
/usr/local/sbin
/usr/local/bin
/usr/bin
/bin
/usr/sbin
/sbin
/Library/TeX/texbin
```

And here is my `$PATH` on the HPC at the University of Arizona:

```
[hpc:login3@~]$ echo $PATH | sed "s:/\n/g"
/rsgrps/bh_class/bin
/home/u20/kyclark/.cargo/bin
/home/u20/kyclark/.local/bin
/cm/local/apps/gcc/6.1.0/bin
/cm/shared/uaapps/pbspro/18.2.1/sbin
/cm/shared/uaapps/pbspro/18.2.1/bin
/opt/TurboVNC/bin
/cm/shared/uabin
/usr/lib64/qt-3.3/bin
/cm/local/apps/environment-modules/4.0.0/bin
/usr/local/bin
/bin
/usr/bin
/usr/local/sbin
/usr/sbin
/sbin
/sbin
/usr/sbin
/cm/local/apps/environment-modules/4.0.0/bin
```

I've used “sed” (or “gsed” which is GNU sed) to add a newline after each colon so you can more easily see that the directories are separated by colons. If you use the `which` command to see where a program lives, you can see that it is found in a directory that is included in your `$PATH`:

```
$ which sed
/usr/bin/sed
$ which python
/anaconda3/bin/python
```

By definition, if a program's location is not in your `$PATH`, then `which` cannot find it, and that means you cannot run it without giving a full path to the program, e.g., `/usr/sbin/foobar/baz`.

In this repository I have included a `bin` directory that has some useful Python programs like `new_py.py` which we will use later to stub out new Python

programs. In order to use them, you have three options:

1. Use the complete path to the programs. E.g., if you have installed this into `$HOME/ppds`, then execute `$HOME/ppds/bin/new_py.py`
2. Copy the contents of the `bin` directory to one of the other directories that are already in your `$PATH`, e.g., `cp $HOME/ppds/bin/* /usr/local/bin`, but that might require root privilege that you don't have.
3. Add `$HOME/ppds/bin` to your `$PATH`

```
export PATH="$HOME/ppds/bin:$PATH"
```

You just told your shell (bash) to set the `$PATH` variable to `$HOME/ppds/bin` plus whatever is was before. Since we want this to happen each time we log in, so we can add this command to `$HOME/.bashrc`:

```
echo "export PATH=$HOME/ppds/bin:$PATH" >> ~/.bashrc
```

As you find or create useful programs that you would like to have available globally on your system (i.e., not just in the current working directory), you can create a location like `$HOME/bin` (or my preferred `$HOME/.local/bin`) and add this to your `$PATH` as well. You can add as many directories as you like (within reason).

Dotfiles

“Dotfiles” are files with names that begin with a dot. They are normally hidden from view unless you use `ls -a` to list “all” files. A single dot `.` means the current directory, and two dots `..` mean the parent directory. Your “`.bashrc`” (or maybe “`.profile`” or maybe “`.bash_profile`” depending on your system) file is read every time you login to your system, so you can remember your customizations. “`Rc`” may mean “resource configuration,” but who really knows?

After a while, you may wish to collect your dotfiles into a Github repo, e.g., <https://github.com/kyclark/dotfiles>.

Aliases

Sometimes you'll find you're using a particular command quite often and want to create a shortcut. You can assign any command to a single “alias” like so:

```
alias cx='chmod +x'
alias up2='cd ../../'
alias up3='cd ../../../../'
```

If you execute this on the command line, the alias will be saved until you log out. Adding these lines to your `.bashrc` will make it available every time you

log in. When you make a change and want the shell to bring those into the current environment, you need to **source** the file. The command `.` is an alias for **source**:

```
$ source ~/.bashrc
$ . ~/.bashrc
```

Permissions

When you execute `ls -l`, you'll see the "long" listing of the contents of a directory similar to this:

```
-rwxr-xr-x  1 kyclark  staff    174 Aug  9 20:21 abs.py*
drwxr-xr-x 14 kyclark  staff   476 Aug  3 12:14 anaconda3/
```

The first column of data contains a wealth of information represent in 10 bits. The first bit is:

- "-" for a regular file
- "d" for a directory
- "l" for a symlink (like a shortcut)

The other nine bits are broken into sets of three bits that represent the permissions for the "user," "group," and "other." The "abs.py" is a regular file we can tell from the first dash. The next three bits show "rwx" which means that the user ("kyclark") has read, write, and execute permissions for this file. The next three bits show "r-x" meaning that the group ("staff") can read and execute the file only. The same is true for all others.

When you create a file, the normal default is that it is not executable. You must specifically tell Unix to change the mode of the file using the "chmod" command. Often it's enough to say:

```
$ chmod +x myprog.sh
```

To turn on the execute bits for everyone, but it's possible to have much finer control of the permissions. If you only want user and group to have execute, then do:

```
$ chmod ug+x myprog.sh
```

Removing is done with a "-", so any combination of `[ugo] [+ -] [rwx]` will usually get you what you want.

Sometimes you may see instructions to `chmod 775` a file. This is using octal notation where the three bits "rwx" correspond to the digits "421," so the first "7" is "4+2+1" which equals "rwx" whereas the "5" = "4+1" so only "rw":

user	group	other
r w x	r w x	r w x

```

4 2 1 | 4 2 1 | 4 2 1
+ + +   + + +   + - +
= 7     = 7     = 5

```

Therefore “chmod 775” is the same as:

```

$ chmod -rwx myfile
$ chmod ug+rwx myfile
$ chmod o+rw myfile

```

When you create ssh keys or config files, you are instructed to **chmod 600**:

```

user    group    other
r w x   r w x   r w x
4 2 1 | 4 2 1 | 4 2 1
+ + -   - - -   - - -
= 6     = 0     = 0

```

Which means that only you can read or write the file, and no one else can do anything with it. So you can see that it can be much faster to use the octal notation.

When you are trying to share data with your colleagues who are on the same system, you may put something into a shared location but they complain that they cannot read it or nothing is there. The problem is most likely permissions. The “uask” setting on a system determines the default permissions, and it may be that the directory and/or files are readable only by you. It may also be that you are not in a common group that you can use to grant permission, in which case you can either:

- politely ask your sysadmin to create a new group OR
- **chmod 777** the directory, which is probably the worst option as it makes the directory completely accessible to anyone to do anything. In short, don’t do this unless you really don’t care if someone accidentally or maliciously wipes out your data.

File system layout

The top level of a Unix file system is “/” which is called “root.” Confusingly, there is also an account named “root” which is basically the super-user/sysadmin (systems administrator). Unix has always been a multi-tenant system ...

Installing software

Much of the time, “bioinformatics” seems like little more than installing software and chaining them together with scripts. Sometimes you may be lucky enough

to have a “sysadmin” (systems administrator) who can assist you, but most of the time you’ll find yourself needing to take care of business yourself.

My suggestions for installing software are (in order):

Sysadmin

Go introduce yourself to your sysadmins. Take them to lunch or order them some pizza or drop off some good beer or whiskey. Whatever it takes to be on good terms because a good sysadmin who is responsive to your needs is an enormous help. If they are willing to install software for you, that is the way to go. Often, though, this is a task far beneath them, and they would expect you to be able to fend for yourself. They may provide `sudo` (<https://xkcd.com/149/>) privileges to allow you to install software into shared locations (e.g., `/usr/local`), but it’s more likely they would expect you to install into your `$HOME`.

Package managers

There are several package management systems for Linux and OSX including `apt-get`, `yum`, `homebrew`, `macports`, and more. These usually relieve the problems of software compatibility and shared libraries. Unless you have `sudo` to install globally, you can configure to install into your `$HOME`.

Binary installations

Quite often you’ll be happy to find that the maintainers of the software you need have gone to the trouble to build binary distributions for your system, which is likely to be a generic 64-bit Linux platform. Often you can just download the binaries and put them into your `$PATH`. There is usually a “README” or “INSTALL” file that will explain exactly what to do. To use the binaries, you can:

- 1) Always refer to the full path to the binary
- 2) Place them into a directory in your `$PATH` like `$HOME/.local/bin`
- 3) Add the new directory to your `$PATH`

Source installations

Installing from source usually means downloading a “tarball” (“tar” = “tape archive,” a container of files, that is then compressed with a program like “gzip” to create a “.tar.gz” or “.tgz” file extension), running `./configure` to figure out how it can build on your system, and then `make` to build the binaries. Usually

you will run `make install` to put the binaries into their proper directory, but sometimes you just `make` and copy the files yourself.

The basic steps for installing into your `$HOME` are usually:

```
$ tar xvf package.tgz
$ ./configure --prefix=$HOME/.local
$ make && make install
```

When I'm in an environment with a directory I can share with my team (like the UA HPC), I'll configure the package to install into that shared space so that others can use the program. When I'm on a system like “stampede” where I cannot share with others, I'll usually install into my `$HOME/.local` or some sort of “work” directory.

The (Data and Software) Carpentries

The Software Carpentry project aims to teach basic command-line usage. You should definitely look through <https://swcarpentry.github.io/shell-novice/>.