

# Unix

## What is Unix?

---

Unix is an operating system, created during 1969 - 1971 by AT&T employees at Bell Labs.

In 1972, Unix was rewritten in the C programming language. It was programmed in assembly before. This allowed it to be portable.

C was developed for the Unix OS.

Because of a court case, AT&T was forbidden from entering the software market, so while they weren't allowed to sell software, they were allowed to give it away for free.

That was very attractive to government, universities and corporations, because if they wrote to AT&T, they'd get a licence and the source code of the OS for free.

The most important of these were universities, because programming really began to take off in 1975, and when new students came to study computer science, the first thing they learned was Unix, so when they went on to found software companies in the 80's and 90's, they based a lot of what they made in Unix.

After 1977, there were a lot of branches and improvements made to Unix. Some of these include:

- Open Source: BSD (Berkley Software Distribution), Linux
- Closed Source: Solaris (Sun/Oracle), AIX (IBM),
- Mixed Source: Mac OS X (Apple)

These days, Unix means a unix-like system, because no-one really uses System 5 anymore.

Mobile devices (iPhone, iPad, Android) are all based on Unix-like systems.

## Mac OS X

---

Mac OS X is a combination of BSD Unix and code from NeXT (a company that Steve Jobs founded after leaving Apple - the code was called NeXTSTEP). Then Apple went and bought NeXT and added more code on top of it, and called the combination of those 3 **Darwin**.

Darwin is the unix that sits underneath Mac OS X. Mac OS X includes a lot more than that - it includes the Finder and other tools too, but Unix is "under the hood".

We can access Unix (aka Darwin) directly from the command line using Terminal.

## Why Use Unix Directly via Terminal?

---

If we have the GUI, why use the command line tool via the Terminal?

- There's a lot more power in the command line
- When working with data, non-data elements can get in the way

## Other Languages

---

Most of what we cover in this course is going to apply to any flavor of Unix. All Linux distributions, Red Hat, Ubuntu, DBN, Solaris, AIX, etc. While there might be small differences in these other OS's, the majority of the knowledge should be the same.

## Using the Terminal

---

The terminal provides command line access to Unix, and comes pre-installed with Mac OS X.

It's located inside /Applications/Utilities/Terminal.

If you're in the Desktop,

```
Shift + Command + u
```

will open the utilities folder.

## Clearing Scrollback

To clear anything above the screen you're in, use View -> Clear Scrollback, or

```
Cmd k      //clears scrollback
```

## What happens when we launch Terminal

The first line tells us the last time we logged in to Unix.

Every time we open a new window, it logs us in to Unix again as a new Unix session. We can have several of those open at once (i.e. several different Unix sessions open at once). Mac automatically logs you in, but other Unix systems don't necessarily log you in automatically. They might require you to put in your username and password to know who you are, but your Mac already knows who you are.

Unix is fundamentally a multi-user environment.

So when you log into your Mac, you're being logged in as a user, so when you open up a Terminal window, it uses those credentials to automatically log you into Unix.

## Commands From the Command Prompt

**echo:** You can use echo to simply print something to the screen.

```
echo 'hello world'
```

**exit:** You can use exit to log out of your session, after which you won't be able to

type anything in.

```
exit
```

**Ctrl - a:** Will move your cursor to the start of a line.

**Ctrl - e:** Will move your cursor to the end of a line.

**Option + Click:** On a Mac, this will take your cursor to a point on the line.  
(Terminal only)

**Tab:** Auto-complete a command

**Tab + Tab:** If it can't auto-complete, this will show a list of possible matches.

**Command + ~:** Will cycle between Terminal windows (Terminal only)

**Command + k:** Will clear screen and clear scrollback (Terminal only)

## Command Structure

---

There is a very definite structure to the commands in Unix.

```
Command      Options      Arguments
```

You can't put the Options after the Arguments. They have to go in the middle.

The Command is always a single word. It's the thing we want to do.

Options will modify the behavior of the command in some way.

Arguments are the thing we want the Command to do when it does whatever it does.

e.g.

```
echo 'hello world'
```

`echo` is the command, and `'hello world'` is the arguments.

An example with an option would be:

```
echo -n 'hello world'
```

The `-n` option suppresses a new line.

Options usually have hyphens in front of them.

```
ruby -v
```

This will give us the version of Ruby we have running. `-v` in Unix usually means - show me what version of this I've got running.

That's the same as `--version`.

Usually you'll have a single dash `-` followed by a letter, or two dashes `--` followed by a word.

You can have single letter options being separate with their own hyphens, or you can combine them all with only 1 dash and no spaces. e.g.

```
ls -l -a -h desktop  
ls -lah desktop
```

There is an exception to this where sometimes an option wants an argument of its own. e.g.

```
banner -w 50 'hello world'
```

The `banner` command has a width option which needs to know how wide you want the banner to be.

So here, 50 is an argument for `-w`, and `'hello world'` is the argument to `banner`.

To make it clearer though, you can remove the space between the option and its argument, so it can also be written:

```
banner -w50 'hello world'
```

We can also have multiple arguments, such as

```
cat -n 3rd.html 4th.html
```

Cat will print out what's in the file, and the `-n` option will print out line numbers.

## Single Quotes or Double Quotes

---

You can use single or double quotes. It doesn't matter to Unix.

## Using Semi-Colons to Break Up Commands

---

Like in regular programming, you can break up commands, even if they are on the same line, with a semi-colon `;`, and they will run one after another. E.g.

```
echo -n "hello"; echo "world"
```

## Kernel and Shells

---

**The Kernel** is the core of the Operating System in Unix. It's what takes care of allocating time and memory to programs. It manages how programs do their thing. Mac OS X uses the Mach Kernel inside Darwin to do that.

**The Shell** is the outer layer of the Operating System. That's what we see when we open up a terminal window. We're working in the shell. It interacts with the user and we can think of it as our working environment. The Shell will send requests to the Kernel, and results will then be returned to the shell.

Mac OS X uses the Bash Shell, but it includes other choices as well.

These options are all available in Mac OS X, and date back to the 1970's, and include:

- sh: Thomson Shell - 1971 //deprecated & replaced by Bourne Shell.
- sh: Bourne Shell - 1977
- csh: C Shell - 1979 (programmer humor)
- tcsh: Tabbed C Shell - 1979
- ksh: Korn Shell - 1982
- bash: Bourne-Again Shell - 1987
- zsh: Z Shell - 1990

For beginners, the difference between these is tiny. Stick with bash to learn, because you are working in the bash Shell when you log in.

We can switch into another Shell even while we are in a Shell. It's just another working layer.

```
echo $SHELL
```

Will tell you which Shell will be used to log you in. Dollar sign `$` and all capitals means that it's an environment variable.

At the moment it reads `/bin/bash`, but that's actually changeable in the Terminal preferences in Startup.

To check the Shell that we're working in right now, we use:

```
echo $0
```

## Changing Shells

To change Shells, just type in the shortened name of that shell, e.g.

```
tcsh
```

Will move us into the Tabbed C Shell (tcsh). If you go a couple of layers deep and

keep changing Shells, to get back, just keep typing `exit` until it tells you you're back at `bash` .

## Unix Manual Pages

---

Unix manual pages help you figure out what you want to do when you're working in Unix.

The manual pages are often referred to as just *man pages*, because the syntax for calling them is:

```
man <command>
man echo
```

If you get a colon at the bottom of the screen, it means that there's more to come. To go forward a page, use `spacebar or f` , and to go back a page, use `b` . To quit, use `q` .

`man` even has its own page - `man man` . There's also a shortened version which is `man -h` or `man --help` .

`man` also has a `-k` option which is the same as `apropos` . This searches the `whatis` database for strings. It searches a set of database files containing short descriptions of system commands for keywords and displays the result on the standard output.

Basically, it gives you short descriptions of system commands. E.g.

```
man -k banner      //returns: print large banner on printer
```

If you don't use the full word, it gives you all the possible options.

We can also use

```
whatis banner
```



And it will print out the same thing. The only difference is that `whatis` doesn't do keyword searching, so if you do `whatis ban`, it will return nothing.

## Directories & Files

---

The working is the directory where we are right now. When you issue commands, it's important to know what directory you're in, because that's where they'll happen.

To find your current working directory, use

```
pwd
```

Which stands for present working directory.

```
ls
```

Will list the files and directories in our present working directory.

```
ls -l -a -h
```

`-l` will show you a long listing, meaning files will be shown vertically stacked.

`-a` will show us hidden files, meaning files that begin with a dot `.` E.g. `.git`

`-h` will return the size of the files and directories in human readable terms.

Dot files, e.g. `.git` is an invisible config file.

`.DS_STORE` is for the desktop/finder to store information about how we're viewing this folder. Window size, which way we're viewing it, etc.

`.bash_history` is a history of the commands we've been typing.

Regarding the results from `ls`:

```
drwxr-xr-x    8 bengrunfeld  staff    272 May  3 15:38 .
drwx-----+ 10 bengrunfeld  staff    340 May  2 15:19 ..
drwxr-xr-x   13 bengrunfeld  staff    442 May  3 15:36 .git
-rw-r--r--@   1 bengrunfeld  staff  64326 May  2 08:49 git_note
s.md
-rw-r--r--@   1 bengrunfeld  staff   1107 Apr 17 17:12 mou_chea
tsheet.md
-rw-r--r--@   1 bengrunfeld  staff   4938 Apr 17 17:12 mou_full
_help.md
```

The `d` at the beginning of the permissions means that it is a directory. The dash `-` means that it is a file. Also, you can sometimes have an `l` there, which means link, or shortcut.

`cd Library/Books` - when using `cd`, you should leave off the slash `/` at the beginning of the path, because if you put it in, it means that this is an absolute path, starting from the root directory.

`cd /` - will take you to the root directory.

`cd ~` - takes you to your user directory.

`cd -` - takes you to the previous directory that you were in.

## File System Organization

In a typical Unix organization, you have the following directories and folders:

**/** - Root of the Hard Drive

Inside the Root, there is typically the following:

**/bin** - where binaries and programs are stored. These are Unix programs, not applications like Dreamweaver.

**/sbin** - are for system binaries and system programs

**/dev** - are where there are references and files for different devices like Keyboard,

hard drives, mouse, etc

**/etc** - where system configurations go

**/home** - where user home directories go

On most Unix systems when you log in, you'll be placed into a directory somewhere inside of **/home**. Not on Mac, but on most Unix systems.

**/lib** - is a place for storing Libraries of code that are referred to by various programs.

**/tmp** - is for temporary files. Temp files are files that you won't really mind if someone comes in and deletes them

**/var** - is for various files that the system uses.

**/usr** - where the user would put programs, tools and libraries. Not their files, the User's files live in their **/home** directory. These are programs that are for the user.

**/usr/bin /usr/etc /usr/lib /usr/local** - as above

Most versions of Unix will adhere to this structure. They may add a folder or take away a folder, but it they'll mostly be the same.

Mac is different though.

## Naming Files

Rules of Unix File Naming:

1. Maximum of 255 characters
2. Avoid most symbols - e.g. `/ \ * & ^ % $ # < >` and others
3. Use A-Z, a-z, 0-9, period, underscore, hyphen (but don't start with symbols)
4. Mostly use lowercase letters because Unix is case-sensitive, although not on a Mac. But it's still an issue if you're trying to pull files on to your Mac from a regular Unix system
5. Underscores are better than spaces

6. escape spaces with `\`
7. Use quotes around names with spaces
8. File extensions (.txt, .php, .html) are not necessary but are very helpful
9. Can't name a file `.` or `..` and shouldn't name any file a Unix command, e.g. `echo.txt`

## Creating Files in Unix

There are 3 main ways to create files in Unix.

1. Unix text editors
2. Direct output to file
3. Touch

```
touch filename.txt
```

What `touch` does is that it reaches out to a file, and if it exists, it touches it and updates its access time. If it doesn't exist, it creates it for us.

But if you `touch` a file or directory that exists, it will update the time at which it was accessed last.

## Using Text Editors

---

In the beginning, developers working on Unix used to use a text editor called "ed", which is short for editor, and is not user-friendly.

Then Unix developers came out with something that was a little more user friendly, called `vi` which is short for 'visual editing mode', which was upgraded and called `vim` (vi improved).

Vim is still in use. It's a modal editor, where your fingers stay in the middle of the keyboard.

Another popular editor is `Emacs`, which is short for "editor macros". Its power comes from all the macros that have been developed around it, which allow you to automate a lot of your work

A better editor for beginners was pico (pine composer), which later became nano (1000x larger than pico), which has basic features and is easy to use. Pico was a very simple email program.

If you type pico into your mac, it will put you into nano.

## Reading Files

---

Yes, you can nano to read file, but there are other tools in Unix that are JUST for reading files.

This is important, because you can output the text of a file to different commands and different programs, not just the screen.

Here are some tools to read files:

`cat` - concatenate. `cat` at its simplest level allows you to read files, but the reason it's called concatenate is that it can also join several files together and print them to the screen, hence the term. The problem with `cat` is that it doesn't paginate the results, so you have to scroll up and down.

`more` - prints a file to the screen, and paginates it, but doesn't allow you to go back through the pages.

`less` - is an improvement of `more` that paginates results but allows you to go backwards through the pages. `less` has completely replaced `more`, (the joke is that less is greater than more), so if you type `more` onto a Mac, it will actually activate `less`.

`man` pages use `less`. `spacebar` or `f` goes forward, `b` goes backward, `q` exits. `g` goes to the start of the document. `shift + g` goes to the end.

`less -M filename.txt` - gives you a better prompt, shows you how far you've got through the document and which lines you're viewing

`less -N filename.txt` - shows you line numbers

## Reading portions of files

`head` - displays lines from beginning of a file (default: first 10 lines)

`tail` - displays lines from end of a file (default: end 10 lines)

Useful for peeking at the beginning or end of a file. Good for logs, where newest stuff is at beginning or end of file.

`tail -f` - will follow the file. It won't exit immediately from viewing the file, but will watch for changes to the file and update your screen if anyone saves something else there.

To exit, use `Control + c`

To look at your system log, use:

```
tail -f /var/log/system.log
```

To see the system log, i.e. requests coming into the server, use:

```
tail -f /var/log/apache2/access_log
```

Or to see the error log, use:

```
tail -f /var/log/apache2/error_log
```

## Creating Directories

---

To create directories, use:

```
mkdir directory_name
```

Or to create a directory inside of another directory:

```
mkdir parent/child
```

But if you create a directory 2 levels deep, it will create 2 directories. So in this example, if parent exists, but the other 2 don't, it will create child and grand\_child. You need to pass in the `-p` option, otherwise it won't work. `-p` stands for **parent**.

```
mkdir -p parent/child/grand_child
```

You can also pass in the `-v` option, which stands for **verbose**. Using this will print to the screen a report of which directories were created.

## Copying Files and Directories

To copy a file, use:

```
cp oldname.txt newfile.txt
```

### `cp` Options:

- `-n` - no overwriting
- `-f` - force overwriting (**default**)
- `-i` - interactive overwriting "ask me"
- `-v` - verbose

Copying directories works the same way, except with one difference. You need to use the `-R` option, which stands for copy Recursively down the line until you finish.

```
cp -R dir1 dir2
```

Sometimes `-r` will work as well, but you should really use `-R`.

## Moving and Renaming Files and Directories

To move a file, use:

```
mv filename.txt destination/filename.txt
```

But destination needs to be an existing directory.

You can even use

```
mv filename.txt destination/  
mv filename.txt destination
```

And these will both work.

To rename a file:

```
mv oldfilename.txt newfilename.txt  
mv oldfile.txt destination/newfile.txt
```

To rename a directory:

```
mv old_dir_name new_dir_name
```

This only works if `new_dir_name` doesn't exist. If it exists, it will simply move the directory. But if `new_dir_name` doesn't exist, it will say, "oh, you want to do a rename. ok. BANG!"

### `mv` Options

- `-n` - no overwriting
- `-f` - force overwriting (**default**)
- `-i` - interactive overwriting "ask me"
- `-v` - verbose



## Deleting Files and Directories

When you delete something, it is destructive, meaning that it is destroyed completely. It is not placed in the trash. It is totally destroyed.

To delete a file, use:

```
rm filename.txt
```

I tested it, and you can also delete multiple files this way just by adding more on to the end.

```
rm file1.txt file2.txt
```

To delete a directory, we have 2 options:

```
rmdir dir_name
```

The issue with this is that `rmdir` will only remove directories that are empty.

To delete a directory that is NOT empty, use

```
rm -R dir_name
```

This will destroy everything inside of the directory, plus the directory itself.

## Links and File Aliases

---

Conceptually, links are similar to file aliases that you create in the Mac OS X Finder. They're not the same thing, though.

1. Create a file
2. Go to the Finder and then navigate to `File -> Make Alias` (or `Cmd + L`) or we can `option + command + drag the file` to create the Alias

3. Rename the new file
4. There will be an arrow on the icon in the Finder that tells you it's an Alias

If the file or the alias moves, it still points to the file, no matter where we relocate either.

If you delete the file, it will break the alias.

We can also make aliases of folders the same way.

Aliases are bigger files, because that is the info the Finder uses to keep track of the original file.

If we open the file alias or directory alias in the Finder, it will open the original file. This doesn't happen in Unix.

If we open the file alias in Unix, it will open as Gibberish (compiled code). They are only for the Finder.

Instead, we'll need to use the Unix version of aliases, which are called **Links**.

## Hard Links

To make a Hard Link

```
ln file_to_link.txt hard_link_name.txt
```

Creates a reference to a file in the filesystem.

Does not break if file is moved.

Unlike Aliases in the Finder, the file does not break if the original file is deleted.

This is because both files are pointing to the same storage address on the hard drive. This is different to the Alias in the Finder, which points to the file, instead of the memory address.

The hard link doesn't even have to have a file extension.

## Symbolic Links

Also known as **sym links**. To create a symbolic link, use:

```
ln -s filetolink symlink
```

The difference between hard links and symbolic links is that symbolic links reference the path to the file. Not the file itself.

So sym links are more interested in the directory the file is in.

Will break if moved. Will break if deleted.

If you do

```
ls -la
```

it will appear as

```
`symlink -> filetolink.txt`
```

## Searching for Files and Directories

---

Spotlight is the searchbar that the finder uses. But here's the Unix way:

```
find path expression
```

e.g.

```
find ~/Documents -name "myimage.png"
```

This will return anything in **Documents** that has exactly the name **myimage.png**.

To search for something less specific, we need to use wildcard characters. In Unix, these are:

```
*    //zero or more characters (glob)
?    //any one character
[]   //any character in the brackets
```

Here is an example of how to use a wildcard

```
find ~/downloads -name '*.csv'
find ~/Documents -name 'git_notes.[p]??'
find ~/Documents -name 'git_notes.*'
```

And there are many other options you can feed in. If you don't want something that has a certain path, use

```
find ~/Documents -name 'git_notes.*' -and -not -path *notes*
```

## File Ownership and Priveledges

---

Because Unix is designed to be a multi user system, it would not make sense if all users has all permissions to everyone elses files.

To find out who you are, use:

```
whoami
```

This is useful if you are logged into a remote user, or if you switch users, etc.

If for some reason your permissions are denied, you can check which user you are, just to be sure.

Each user gets a home directory. `cd ~` to get to it.

That value is stored in `$HOME`

You can create other users through OS X interface, but we prefer to do it from the command line.

## Unix Groups

A **Group** in Unix is a set of users. Each user belongs to at least 1 group. A primary group. And they can belong to any number of other groups as well.

Groups are good for associating a group of users with a file. So file permissions can be set by group. (e.g. IT Group), so all you'd have to do is add a user to a group and they'd have access to that file.

Groups are really used for shared systems, like remote servers, etc.

To see which groups you belong to, use:

```
groups
```

## File and Directory Ownership

Ownership is an essential part of working in a multi-user environment. It's how Unix determines which files you can access and which ones you can't. You can see the ownership of files and directories everytime you use `ls -la`

It's the second and third columns in the results list. E.g.

```
drwxr-xr-x    6 root          admin      204 Jan 17 12:53 ..
-rw-----    1 bengrunfeld  staff        3 Jan 26 09:11 .CFUser
TextEncoding
-rw-r--r--@   1 bengrunfeld  staff    15364 May  6 08:25 .DS_Sto
re
drwx-----   20 bengrunfeld  staff     680 May  8 09:09 .Trash
```

The owner for almost all of the files above is `bengrunfeld` .

The group that owns most of the files about is `staff`

We can set permissions based on the owner or based on the group.

We can share files with other users.

To change the permissions of a file, we use

```
chown user:group filename.txt
```

which stands for "**change ownership**".

To change only the owner or only the group, use:

```
chown user filename.txt  
chown :group filename.txt
```

We can do this for directories as well, but it won't change all of the contents of the directory. The files inside will keep the ownership they already have. To change everything in the directory as well, we use:

```
chown -R user:group filename.txt
```

`-R` stands for recursively. This will go down through everything inside the directory.

You can't just change the ownership to another user, since that would be a security concern. You can only do this if you are an administrator on the machine, and then you need to use `sudo`. E.g.

```
sudo chown user:group filename.txt
```

Then it will ask for your password to make sure.

## File and Directory Permissions

To see permissions we use `ls -la`.

```

drwxr-xr-x    6 root          admin    204 Jan 17 12:53 ..
-rw-----    1 bengrunfeld  staff      3 Jan 26 09:11 .CFUser
TextEncoding
-rw-r--r--@   1 bengrunfeld  staff  15364 May  6 08:25 .DS_Sto
re
drwx-----   20 bengrunfeld  staff    680 May  8 09:09 .Trash

```

We see permissions in the first column, where the first character indicates if it is a file, directory, or link. So, `d` for directory, `-` for file, `l` for symbolic link.

The next 9 characters after that are an indicator of the permissions for this file or directory.

This system is called **Alpha Notation**

We have 3 groups: User, Group, Other

User is us, Group is whatever group we belong to, and Other who is everyone else who might have access to this file.

For each group, we can have 3 levels of permissions: Read, Write and Execute.

Read means we can read the contents of the file. Write means we can change the contents of the file. Execute for a file means that we can run it, but for a Directory, it means that we can search inside of it.

To represent the permissions, we use

```

read (r)
write (w)
execute (x)

```

Here's how to notate the permissions that you'd like to grant:

```

rwx
rw-
r--

```

If you put them all together, you get 9 characters, which is what you see with

```
ls -la
```

Usually for text files, you don't enable execute permission, because it's not a script. You only need to read it.

By default, directories are given execute permission when they are created.

## Changing Permissions for Files and Directories

We use `chmod` to change permissions - which stands for **Change Mode**, which is how Unix refers to the mode of these files (i.e. permissions).

```
chmod mode filename.txt
```

We're going to use U, G, and O, to stand for User, Group, and Other.

To give ALL groups ALL permissions, use:

```
chmod ugo=rwx myfile.txt
```

To set different permissions for different groups, use:

```
chmod u=rwx,g=rw,o=r myfile.txt
```

To change the permissions to something slightly different without having to write all that out, use:

Give User and Group write permissions to this file:

```
chmod ug+w myfile.txt
```

Or to take away permissions, use:

```
chmod o-w myfile.txt
```



A shorthand notation to do all 3 groups together. Instead of:

```
chmod ugo+rw myfile.txt
```

Use:

```
chmod a+rw myfile.txt
```

which has the same effect.

If we `chmod` a directory, it will only change the directory itself, not all the files that are in it. To recursively change all the files in the directory, and all of its sub-directories, use:

```
chmod -R g+x dirname
```

## Changing Permissions with Octal Notation

Octal notation is very popular and a lot of people use it.

Instead of using letters, **Octal Notation** uses numbers.

```
r = 4  
w = 2  
x = 1
```

Here's how to use Octal Notation

.	User	Group	Other
Read (r)	4	4	4
Write (w)	2	2	0
Execute (x)	1	0	0
<b>Total:</b>	7	6	4

So the total when you add them up is:

Those 3 digits (764) are the Octal notation of the 9 rwx characters. Hence:

```
rw-rw-r-- = 764
```

So to give a file ALL permissions using Octal notation, use:

```
chmod 777 myfile.txt
```

To give rw-rw-r-- permissions, use:

```
chmod 764 myfile.txt
```

A popular combination is:

```
chmod 755 myfile.txt
```

This means that I (User) have all privileges, but everyone else (Group + Other) only have read and execute permissions.

```
chmod 000 myfile.txt
```

Means that no-one has ANY privileges, but a more common way of doing this is by giving the User privileges, but no-one else, so:

```
chmod 700 myfile.txt
```

## The Root User

The Root User is a super-user account that can do **ANYTHING** on the Unix system.

It's an all-powerful user that can read, write, or execute anything, delete user accounts, change permissions, etc.

The Root User is disabled by default on Mac OS X after it has set everything up for you. Recommended NOT to re-enable it, because there's another way.

Remote Unix servers usually do have Root enabled.

## Sudo

Sudo is a command that allows us to temporarily take on the powers of the Root user.

Even though the Root User is disabled on the Mac, as Admin users, we can do everything that the Root User can do. We just have to do it using the `sudo` command.

`sudo` stands for **Substitute User and Do**.

Some people mistakenly believe that it stands for **Super User Do** because the Root User is a super user, but what it's actually doing is substituting in a different User identity.

`sudo` is a command that prefixes other commands.

E.g.

```
sudo ls -la
```

This just means, do `ls -la` as the Root User.

It will then ask for a password, just to make sure.

Once you've entered your password, it won't ask for your password again for about 5 minutes, even if you use the `sudo` command again. This can be changed in your configuration file.

To expire the password right now, use:

```
sudo -k
```

## Changing Users

Root is not the only User you can become.

```
sudo whoami
```

Will tell you that you're Root

```
sudo -u greg whoami
```

This will return greg (assuming greg is a User on your machine).

We're substituting our identity before we execute the command. So we can become a different User and take on THEIR priveledges and THEIR role as easily as we can root.

So I could look at, write to, or read their files AS THEM.

Not everyone can do `sudo`. If you go into settings in the GUI -> Settings -> Accounts, you'll see some Users are Admins and some aren't. Being an Admin means you can use `sudo`. If you're not an admin, you can't use it.

In Unix, there's something called a `sudoers` file. To get to it, use:

```
sudo cat /etc/sudoers
```

The bottom lines set which Users and which Groups have `sudo` access, aka admin access.

## Unix Command and Programs

---

Every command is a program, but we call them commands for convenience. Commands are files that are executable that get executed when you run them.

Command files are stored in `/bin` in Unix, so for example, `echo` is stored in `/bin/echo`.

To execute a file, you just type the filename and hit `return`. This is a shortcut for typing `/bin/echo "hello there!"`

Commands are just a shorthand for saying 'go to these files and execute them'.

To find where the file of a command is, use either of:

```
whereis echo
which echo
```

We can also use

```
whatis echo
```

but that will return much more information.

## Command Basics

Most of the time (but not always), you can pass in the following options successfully:

```
-v
--version
--help
```

To exit out of a program, use one of the following:

```
q
x
Control + q
Control + x
ESCAPE
```

Or if you're really stuck, you can perform a FORCE QUIT with

```
Control + c
```

Or you could close the window, but the process might keep running.

We can also put semi-colons `;` between commands to run multiple commands on one line.

## The PATH Variable

`echo` is just a shortcut to `/bin/echo`. But if I create another file called `echo`, how does Unix know which one to use?

Unix manages all this by using a variable called `$PATH`.

`echo $PATH` will show you the current value of your Path. i.e.

```
/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/local/git/bin
```

The Path is always a colon-separated list of file paths and this is the list that Unix will use when trying to locate commands to execute. The early entries take precedence over the later ones, so what it does is that it starts at the left, and when we type `echo`, it says, "I'll start in `/usr/bin`". If it doesn't find `echo` there, it will look in `/bin`. In this case, it will find `echo` there, but if it didn't, it would then keep looking in the file paths that come after it.

If it didn't find it in any of those places, it would say that the command could not be found.

This is what happens if you just type in some gibberish like `fstenr`. It will look in all of those paths and then say that it could not find it.

We can change this in **bash** by just typing

```
PATH=<whatever you want the path to be>
```

With Web Development, we'll usually have a `mysql` folder, which we want Unix to look in when we type a MySQL command.

Setting the Path like this only lasts for the current session. So if you close the window and then open it again, whatever you saved into Path will have been reset.

If we use `which ls`, it will tell us which version of `ls` we're using. There may be several versions of `ls` that are there, but it tells us the Path variable as a way of telling us which version it will use.

`which` is slightly different than `whereis` because `which` uses our path to tell us which one is active.

## System Information Commands

To see the current date, use

```
date
```

Remember, this is the date that the computer has currently been set to, which may not be entirely correct if it has been set to something else. Same as on clock in GUI.

```
uptime
```

Will tell you how long the computer has been on for, and returns something like this:

```
11:42 up 2 days, 17:45, 2 users, load averages: 2.09 1.98 1.86
```

Here you can see that there are 2 users. If you type:

```
users
```

It will show you what users are there, but will only return 1 User. To see the other User, you'll need to use:

```
who
```

this shows us a list of ALL users and what they're doing, and it doesn't remove duplicates. `users` DOES remove duplicates.

One of these Users is the terminal window, and the other is the GUI of the operating system (console).

```
uname
```

Will return the operating system name. This is the Unix name. OS X runs on top of that.

```
uname -mnrsvp
```

Will give you a heap of information about the operating system. A shortcut for this command is to use `-a` which covers everything besides `-p`, so you still have to add it in there.

```
uname -ap
```

For network environments (e.g. on a remote server): To find the name of the host that you're on, use:

```
hostname
```

And to find the domain name, use

```
domainname
```

## Hard Drive Information Command (aka Disk Info Commands)

```
df
```



Stands for **disk freespace**, and will return the amount of disk freespace available to us.

There are a couple of devices, but usually we ignore these. We look for the things with the big numbers.

Because these numbers can be a bit hard to read, we have the `-h` option, which stands for **humanize**.

```
df -h
```

This will display memory units in kilobytes, megabytes, and gigabytes. There is an issue with this, though. Memory units can be calculated in base 2 and base 10.

To use base 10, use `-H`

```
df -H
```

This will show us the bigger number.

To see Disk Usage, use:

```
du
```

We need to provide it the path that we want it to look at, and it will give us a lot of information about the disk usage at that path.

We wouldn't want to do this at the root of our hard drive because it would tell us every single file and folder on the hard drive.

```
du ~/Documents
```

This will tell you what the size is for each of the directories that it sees.

For the Human Readable version of the memory units, use `-h` . e.g.

```
du -h ~/Documents
```

It has a `-H` option, but it is totally unrelated to Human Readable output, so you wouldn't want to use it.

To see all of the files as well as the directories, use the `-a` option.

```
du -ah ~/Documents
```

To tell Unix how many directories deep it needs to go, use the depth option `-d`

```
du -hd 1 ~/Documents
```

This will only return results 1 directory deep. If we wanted to see 2 directories deep, we'd use 2. etc. 0 means just this directory.

`du` returns the size that has been set aside for these files on the hard drive. That's different from the size the files are actually using.

To see the actual size the files are using, use:

```
ls -lah ~/Documents
```

and you'll see that most of the time the files use much smaller amounts than what is returned with `du`. The reason for this is because the file systems sets aside **blocks** of space for files, even if the file doesn't take it all up. There is a minimum size for that. `du` returns the block size, not the actual size.

This is what defragmenting the hard drive does. It reclaims size by changing the block size to be much closer to what the actual file size is.

## Unix Commands for Managing Processes

There is the Kernel and there is the Shell, which in our case is Bash.

Whenever we run a command inside our Shell, a file executes and it communicates with the Kernel. Essentially, it says to the Kernel "there are some things I need to accomplish here. Can you help me out?" The Kernel sets aside

some memory space and starts a process running in it.

Then whenever there's output from that process, it returns it back to the Shell for us to see, and whenever the process is done, the Kernel closes it out and reclaims that memory space so that it can be used by other processes. That's what the Kernel does, it manages the processes for us.

We can have processes that are really short (like the `echo` command), or we could have **longer running processes** (e.g. feeding 5 pages to a printer).

We can also have background processes, e.g. a database server like MySQL, it will just sit in the background waiting for connections and requests, which it will respond to.

To see those processes, use:

```
ps
```

stands for Process Status. It gives us a snapshot of the processes that are running. There are a lot of processes running on the machine, but we don't see these with `ps`. What if we are processes that are owned by me, and processes that have a **controlling terminal**, i.e. that is, I'm in control of them. They are not **background processes**.

To see processes that are owned by other users, use:

```
ps -a
```

If you actually run `ps -a`, it will return `ps -a`, even though we don't own it. Root owns `ps -a`, but we can still see it because it is acting on our behalf.

To get a different implementation of ps, use

```
ps a
```

This no-hyphen version is from a different version of Unix. The reason this is important is because the most classic way to use `ps` is:

```
ps aux
```

`a` means show my all processes regardless of who they're used by, `u` means include a column that shows the user who owns the process, and `x` shows me the background processes too.

This will really show you a list of all the processes you have running right now.

```
USER  PID  %CPU  %MEM  VSZ   RSS  TT   STAT  STARTED  TIME  COMM
AND
```

This is the header information.

`PID` is Process ID. Every process gets assigned a unique ID to help us keep track of it.

Then the percentag of CPU being used `%CPU` . Then the percentage of memory that it is taking up `%MEM` , then the amount of virtual memory that it is taking up `VSZ` , then it shows us if the terminal has launched it `TT` . Question markes under `TT` means that Mac OS X launched it.

`STAT` means status. `STARTED` means the time that it started. `TIME` tells us the amount of time that it's been running, and `COMMAND` is the command itself, i.e. a Path to where the file of the command lives.

Remember that `ps` command only gives you a snapshot of what's going on, and this changes.

## Monitoring Processes

To really watch the process live, i.e. to monitor it, and see the CPU and MEMORY go up and down and processes start and stop, use:

```
top
```

This will show you a list of the top processes. The top processes depends on how

you sorted it, and by default it is sorted by PID.

The most recent processes are at the top. Above the results, you'll see some summary information. To exit out, just use `q`.

To set how many processes will show, use

```
top -n 10
```

This will show only 10 processes. By default it shows how ever many fit into the window you're viewing.

To sort by CPU, use

```
top -o cpu
```

To set the refresh rate (default is 1 second):

```
top -s 3
```

Will set the refresh rate to 3 seconds

To filter by User, use

```
top -U bengrunfeld
```

The `?` key will bring up a help screen.

You can also update options while `top` is running.

If you hit `s` and then `1`, it will set the refresh rate to 1 second. To see ALL users, hit `U`. The only thing we can't change from within the program is the `-n` value - i.e. how many processes are showing.

Linux has a lot more options that you can use with the interactive mode of `top`.

## Stopping Processes

The best way to stop a process is to use `Control + c`. Sometimes we can't use that, though. This usually happens when it's a background process, or the process has gone out of control.

To stop a process, we use the `kill` command. What we need though is the process ID (PID), that's how we tell Unix which process we want to kill.

```
kill PID
```

e.g.

```
kill 1953
```

Some processes can't be killed by using this process.

Try `kill` first, but if it can't do it, use:

```
kill -9 1842
```

This says to Unix "I know better than you! You really need to kill this thing off."

## Text File Helpers

These commands help with text files - which are files that have nothing in them besides text.

`wc` - word count `sort` - sort lines `uniq` - filter in/out repeated lines

If we use `wc`, we get back 3 numbers and then the name of the file:

```
8      7      41 basket.txt
```

The first number is the number of lines in the file. The second number is the number of words in the file. The third number is the number of characters in the file.

Unix defines a word as text with spaces on either side of it.

Unix defines a line as something ending with a line return. So a paragraph with wrap around text wouldn't necessarily constitute a whole bunch of lines.

To sort the lines in a file, use

```
sort basket.txt
```

This won't actually modify the file, it will only sort the output that is printed to the screen.

Usually upper-case letters come before lower case letters (ASCII) in a sort, but we can pass in the `-f` option to tell Unix that we want an `A-Z` sort regardless of upper or lower case.

```
sort -f basket.txt
```

To reverse sort, use `-r`, so:

```
sort -r basket.txt
```

To sort with uniques removed, use the `-u` option:

```
sort -u basket.txt
```

To remove repeated lines, where the repeated line came directly after its copy, use

```
uniq basket.txt
```

To only print the lines that are repeated, use

```
uniq -d basket.txt
```

To see ONLY the un-repeated lines, use

```
uniq -u basket.txt
```

## Utility Programs

`cal/ncal` - Calculator Program `bc` - bench calculator `expr` - expression evaluator: a simple one-line calculator `units` - units of measure conversion

### Calendar

`cal` will give you the current month's calendar

```
cal 12 2020
```

Will give you a calendar of the month of December in 2020.

```
cal -y 1980
```

Will show a calendar of the full year of 1980.

`ncal` is a different program that rotates the calendar so that the days of the week run down the left hand side vertically.

### Bench Calculator

If you enter `bc`, you will go into a program. It's quite a powerful program with it's own maths-related programming language which is similar to C.

You can just enter `1+1` and hit return and it will give you the answer. To quit, type `quit`.

To change the precision re decimal places, use `scale=10` for 10 decimal places.

By default, the precision is 0.

### Expression Evaluator

To calculate something really simple, you can use `expr`. You need to put spaces in between values, so:



```
1 + 1
```

To do multiplication:

```
123 \* 456
```

You need to escape the `*` first.

## Units

Typing `units` will drop you into the units program. It then tells you how many different units of measurement it has, and how many prefixes.

```
You have: 2 feet
You want: meters
* 0.6096
/ 1.6404199
```

I entered in 2 feet, and then meters, and it tells me that that 2 feet is equal to 0.6096 meters. Then it gives us the reverse calculation 1.6404199, because this is sometimes more precise.

For temp:

```
You have: 60 degF
You want: degC
```

To quit, use `Control + c`.

We can also use `units` from the command line without dropping into the program. E.g.

```
units '2 miles' 'meters'
```

## Unix Command History

Unix remembers our previous commands (e.g. when you use the up arrow). Even if you close the Terminal window and even reboot your machine, it will still remember your last command.

This history is stored in a file at `~/.bash_history`

Bash only writes your old commands to `bash_history` when it quits, so right now it won't have your last few commands made from the current window. Until we quit, it just holds onto the current commands in memory.

To really see all commands, including the ones not yet written to `bash_history`, use:

```
history
```

The list that comes back is numbered, and we can use those numbers to do the command again. Using these numbers is easier than hitting the up arrow, so if something was made 200 commands ago, it's better to do it this way.

To run an old command when you have its number, use:

```
!5
```

for the command with the history id of 5.

If you then hit the up arrow, it will bring up the last command itself, not `!5` which is a shortcut. If you then edit it and press the down arrow, then run `history`, it will show you the edited version with an asterisk next to it to indicate that the command was edited.

To tell Unix that you want to run a command from 2 commands ago, use:

```
!-2
```

This will run edited commands.

If you ran `expr 32 + 62` a few commands ago, you can run that exact

command again by typing:

```
!expr
```

This will look for the last `expr` command you ran and then run it again. You could even type `!ex` and it will still work.

A way of saying "run the last command" is `!!`. So say you run:

```
chown bengrunfeld file.txt
```

and it throws you an error, you can just run

```
sudo !!
```

Alternatively, `!$` will reference the arguments from the previous command.

To delete a line out of `history`, use:

```
history -d 68
```

where 68 is the line number. Everything will then shift up one.

```
history -c
```

will remove all of your history. Or you can just delete the `.bash_history` file.

## Standard Input and Standard Output

---

### Standard Input

The standard input is the Keyboard. `stdin` `/dev/stdin`

### Standard Output

The standard output is the window or terminal, where it outputs its results to.

`stdout` `text/terminal` `/dev/stdout`

We're going to be changing each of these.

## Directing Output to a File

To direct the output of a command to a file, we use the `>` character. E.g. with `sort`:

```
sort mylist.txt > sortedlist.txt
```

The file `sortedlist.txt` doesn't have to exist yet, as this process will create it on the fly.

Any command that outputs to the screen can be outputted to a file instead.

This command does overwrite in a destructive fashion. It will replace anything inside a file that already exists, if you write to it.

A classic use case for this command is:

```
cat file1.txt file2.txt > joined_files.txt
```

This concatenates `file1` and `file2` and then writes the output into `joined_files.txt`

## Appending Output to the End of a File

To append something to the end of a list instead of completely overwriting an existing file, use `>>` instead of `>`.

```
echo "apple" >> fruitbasket.txt
```

## Directing Input from a File

Instead of using input from the Keyboard, we can use input from a file.

```
sort < fruitbasket.txt
```

This will take the input from the file `fruitbasket.txt` and direct it to the command `sort`. Of course, this doesn't do anything different, because that's what the command does anyway, but this is the general theory.

A clearer way of seeing this work is with `bc`.

```
echo "768+49-(6/2)" > calc.txt  
bc < calc.txt
```

## Combining Input and Output Methods

We can use both `<` and `>` in the same command.

```
sort < fruitbasket.txt > sortedfruit.txt
```

This will input `fruitbasket.txt` to the `sort` command, and then write the output to `sortedfruit.txt`.

Two rules:

1. The input always has to come before the output
2. The arguments to these always have to be files

## Taking the Output of a Command and Using it as the Input to Another Command

To take the output of a command and use it as the input for another command without having to use a file as a go-between, we use:

```
echo "Hey there" | wc
```

This will take the output of `echo "Hey there"`, and use it as the input for `wc`.

Or we can use it like this:

```
echo "(4*7)-6" | bc
```

We use Pipe `|` as a verb, saying we want to pipe something into something else.

You can use pipe `|` multiple times in a single line.

```
echo "(4*7)-6" | bc > calc.txt
```

This can be really useful, like if we wanted a paginated of our processes:

```
ps aux | less
```

Remember, when using pipe `|`, we are piping it into a command - not a file - and the output from it should be from a command - not a file.

## Suppressing Output

There are times when you don't want to output anything to the screen, and you don't really want to output it to a file either.

To suppress the output altogether, we output it to a special file called the "null device", aka the "bit bucket" or the "black hole". It lives in `/dev/null`.

Unix discards any data that is sent there.

```
ls -la > /dev/null
```

## Configuring Bash to Your Personal Preferences

---

To do this, we provide commands inside Bash resource files.

Upon login to a bash shell: `* /etc/profile * ~/.bash_profile, ~/.bash_login, ~/.profile, ~/.login`

When you open a terminal window, or log into a bash shell in any other way, the first thing it does is it reads any commands that are inside `/etc/profile`. Those are master default commands that every Bash user will get. We don't want to mess with those, so we'll leave them alone.

Instead, we want to take care of personal customizations that apply only to us. Because they're personal customizations, they're going to live inside our User directory.

The 4 file paths above all begin with `~`, which means they're all in our User directory. Also, they are all dot files. As a dot file, they are not visible in the finder, and the dot also says that they're really a configuration file.

We have 4 choices. After Bash reads through `/etc/profile`, it goes to our user directory and it sees which one of these 4 files it can find, *in the order above*.

When it finds one, it reads the commands that are in there, and then it ignores the rest of the files. So it just goes until it finds one and then executes only that file. So there's no point in putting commands in both `.bash_profile` and `.bash_login`. Any commands in `.bash_login` will just get ignored in that instance.

We're going to put everything into `~/.bash_profile`.

`~/.bash_profile` and `~/.bash_login` are there for bash. `~/.profile` and `~/.login` are there for historical and backwards compatibility reasons, and they apply to other shells.

If we're using other shells, we're better off using their profiles instead.

That only covers when we open a new shell by opening a new terminal window.

If we type `bash` from the command line, it will open up a new shell, but it's not a login shell. There's a distinction between them. The login shell is the first time when we open up the terminal window, but typing `bash` on the command line opens up a sub-shell.

When we start a new bash subshell, it executes the commands inside of

`~/.bashrc` where `rc` indicates resource.

So anything we put into `~/.bash_profile` isn't available to us when we go into a subshell, and anything we put into `~/.bashrc` isn't available to us *until* we go into a subshell.

So we want to have a way where we have one set of commands that are available to us in both cases. We want the same configuration regardless of whether we're in the master shell or in a subshell.

Upon logging out of a bash shell, Unix will run any command that's inside the `~/.bash_logout` file.

To get the load order for other shells, use:

[http://en.wikipedia.org/wiki/Unix\\_shell](http://en.wikipedia.org/wiki/Unix_shell)

The way to deal with the difference between the shell and the subshell is to write a script that says "if you find `.bashrc` then load it into the main `.bash_profile` file."

Add to `~/.bash_profile` :

```
if [ -f ~/.bashrc ]; then
    source ~/.bashrc
fi
```

Now whatever is in `.bashrc` will be inside of `.bash_profile`. When you load the terminal window, whatever is inside `.bash_profile` and `.bashrc` will run, and whenever you load a subshell, whatever is inside `.bashrc` will run.

## Setting up Command Aliases

This is not a Finder alias. It is a Command alias - which is a shortcut that will execute different commands in bash.



```
alias
```

Will return a list of all the currently defined aliases.

```
alias ll='la -la'
```

This means that anytime in the future that I type `ll` in the command line, it will run `la -la`.

Aliases only last for the current login. To make sure you can use them every time you log into Unix, put the alias definitions in `.bash_profile`.

Another use of `alias` is to redefine some unix commands, but with options.

```
alias mv='mv -i'
```

Another thing some Unix users do is correct their typos ahead of time

```
alias pdw='pwd'
```

## The Source Command

You can use `source` to run the contents of a file.

```
source .test
```

this will run the code inside the `.test` file.

## Setting and Exporting Environment Variables

Environment variables are also known as Shell variables. We use environment variables so that we can configure our working environment.

An example of Shell Variables is:

```
echo $SHELL
```

and it returns the default login shell for the current user. Shell variables are written in all caps. The dollar sign `$` is not actually part of the variable name. The dollar sign `$` is there to serve as an indication to Unix that we want it to return the value that's stored in the Shell variable.

We can also define our own Shell variables as well. e.g.

```
MYNAME='Ben Grunfeld'  
echo $MYNAME
```

Like command aliases, any Shell variables that we set in the current session will be erased once we exit the session. If we want them to always be available, they have to be defined in `.bash_profile` or in `.bashrc`.

Unfortunately, defining it this way means that the Shell variable won't be available to child processes that Bash starts for us. It will only be available in Bash itself. To indicate to Bash that it should also pass along these variables, or export them to other commands, programs and scripts, we need to use the `export` command.

```
MYNAME='Ben Grunfeld'  
export MYNAME
```

Now Bash will know that it needs to export the Shell variable to all of the programs that it runs as well.

While it's kind of useful to set your own variables, it is much more useful to use variables to set configuration options for our Unix environment and the programs we use.

E.g. with Less, there is an environment variable that can be used to configure the default settings that Less will use when it starts up. e.g.

```
export LESS='<options we want less to use>'
```

You can initialize the variable, set it, and export it by just using the `export` command.

```
export LESS='-M'
```

By putting this declaration inside of `.bashrc`, it will be transferred by Bash to all the programs that Bash starts, including Less.

## Setting the PATH Variable

Path is a colon delimited list of file paths that Unix uses when it's trying to locate a command that you want it to run. To see it, use

```
echo $PATH
```

Unix will look in those directories for the command in that order. To set the `PATH` variable use the same as above:

```
PATH= ' '
```

This will essentially unset the `PATH` variable, meaning that no command will work.

So it is best to put the `PATH` variable in the `.bashrc` file, using an export as above, as a starting point.

```
export PATH='/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/local/git/bin'
```

A common change that people make is to change the order which Unix uses to find the command.

You can also use the `PATH` variable to create a new path, like so, but we need to use double quotes `" "`. Single quotes `' '` won't work.:

```
export PATH="/usr/local/bin:$PATH"
```

## Configuring Unix History with Variables

Environment variables can help us configure how the Unix history works. There are 5 different variables that we can configure.

```
HISTSIZE=10000                                # Default is 500
HISTFILESIZE=1000000
HISTTIMEFORMAT='%b %d %I:%M %p '              # Using strftime format
HISTCONTROL=ignoreboth                        # ignoredups:ignorespace
HISTIGNORE="history:pwd:exit:df:ls:ls -la:ll"
```

`HISTSIZE=10000` is the number of commands Unix will remember. When it gets to 10000, the oldest command will drop off the top.

`HISTFILESIZE=1000000` is the maximum file size that the history file is allowed to become. This means 1000000 kb, which is a large amount. When it gets to that file size, the oldest command drops off, so that the file doesn't exceed this size.

`HISTTIMEFORMAT='%b %d %I:%M %p '` provides a timestamp next to each of your history entries, letting you know when they were added.

`HISTCONTROL=ignoreboth`. There are 3 things you can set this to. `ignoredups` means ignore duplicates, i.e., don't record the same line multiple times. `ignorespace` tells history not to record any time that begins with a space. Why would we use this? If you're typing a command that contains sensitive information, like a password, then all you have to do is begin it with a space, and Unix will run it, but History won't record it.

`HISTIGNORE="history:pwd:exit:df:ls:ls -la:ll"` ignores certain commands all the time.

## Customizing the Unix Command Prompt

It's really nice to make the prompt look exactly how you'd like it to. We'll use a

Shell variable to do this - `PS1` . There is also `PS2`, `PS3`, `PS4` etc, but those are for other prompts.

Like other Shell variables, we need to store it in the `.bashrc` file or `.bash_profile` to make sure it stays there. Remember to use `export` as above.

To print out your username, use:

```
PS1='\u '
```

Here are some popular formatting codes:

```
\u      # username
\s      # current shell
\w      # current working directory (full path)
\W      # basename of current working directory
\d      # date in "weekday of month" format (Mon Feb 22)
\D{format} # date in strftime format ("%Y-%m-%d")
\A      # time in 24 hour HH:MM format
\t      # time in 24 hour HH:MM:SS format
\@      # time in 12 hour HH:MM am/pm format
\T      # time in 12 hour HH:MM:SS format
\H      # hostname
\h      # hostname up to first " . "
\!      # history number of this command
\$      # when User ID is root (0), will display "#", otherwise a "$"
\\      # a literal backslash
```

The default Mac setting is:

```
PS1='\h:\W \u$ '
```

## Customizing the Logout File

In your `~/` directory, you can create a `.bash_logout` file that runs

whenever you `exit` the terminal or close the window.

## Unix Power Tools

---

### Searching with Grep

Grep is a very powerful Unix tool. Grep stands for "Global Regular Expression Print". Grep was created for Unix.

To search for a text string inside of a file, use:

```
grep <expression> <filename>
```

e.g.

```
grep banana fruitbasket.txt
```

This will return the entire line which contains the expression you're searching for. Grep is REALLY fast.

When used like this, Grep is case sensitive.

Grep Options:

```
-i      # makes Grep case insensitive
-w      # match only on whole words
-v      # return lines the DON'T match the expression
-n      # returns matches with line numbers
-c      # count option: returns a number of how many matches t
here were
```

### Grepping for Multiple Files & Other Inputs

To check inside of a directory, use the recursive option `-R`.

```
grep -R banana food/fruitbox/
```

This will look in the directory and in sub-directories as well.

Will return matches it made, plus the files where it occurred.

```
-h      # will suppress the file name, and just show you the line itself
-l      # will just show you the file name, but will suppress the path
-L      # gives you file names that did NOT contain a matching expression
```

To search inside files matching a regex:

```
grep banana *fruit.txt
```

To use input from another file:

```
cat fruitbox.txt | grep apple
```

To use piped input in a more useful way, use:

```
ps aux | grep terminal
```

This will take everything that gets outputted from `ps aux` and only print to the screen things containing `terminal`.

So to see all of your applications that are running:

```
ps aux | grep applications
```

Or

```
history | grep unix_files
```

Or

```
history | grep nano | less
```

will give us a paginated list of all the times stored in our history that we've used nano.

## Coloring matching expressions

```
grep --color banana fruitbox.txt
```

This will color any matching expressions.

```
grep --color=auto banana fruitbox.txt
grep --color=always banana fruitbox.txt #always uses color
grep --color=never banana fruitbox.txt  #never uses color
```

`auto` says "color it if you're showing it to me on the terminal but not when you're sending it somewhere else like to a file or a pipe." This is because special formatting is used to create the colors around phrase matches, and if you export this to a file using `always`, it will copy the colors across. `auto` strips the formatting if you are exporting.

The default color of phrase matches is red, but we can change that with a Shell variable. Add this to `.bashrc` or `.bash_profile`.

```
export GREP_COLOR="34;47"
```

We can also set other options in the `.bashrc` file, e.g. color should always be turned on, and Grep should always be case insensitive:

```
export GREP_OPTIONS="--color=auto -i"
```

## Grep Color Codes



Attributes	Text Color	Background
0 = reset	30 = Black	40 = Black
1 = reset	31 = Red	41 = Red
2 = reset	32 = Green	42 = Green
3 = reset	33 = Yellow	43 = Yellow
4 = reset	34 = Blue	44 = Blue
5 = reset	35 = Purple	45 = Purple
6 = reset	36 = Cyan	46 = Cyan
7 = reset	37 = White	47 = White
8 = reset		

## Using Regular Expressions

In general when working with Regex, it's a good idea to put quotes around the expression you're searching for:

```
grep 'banana' fruitbox.txt
```

The reason for that is you use special characters in Regex that have their own meaning to Unix, so by putting them in quotes, you're making sure that no confusion occurs.

When you search for `banana`, you're only searching for those exact characters.

A one character wildcard is the dot character `.` so if you had:

```
grep 'b..ana' fruitbox.txt
```

Any 2 characters could go there and it would return a match.


```
grep 'app[lr]' fruitbox
```

The square brackets `[]` means one of the characters in the square brackets, so the above grep would return `apple` as well as `appropriate`

## Regex Expression Basics

Regex	Meaning	Example
.	Wild card, any one character except line breaks	gre.t
[]	Any one character listed inside []	gr[ea]y
[^]	Any one character NOT listed inside []	[^aeiou]
-	Range Indicator (only when inside a character set)	[A-Za-z0-9]
*	Preceding element can occur zero or more times	file.*
+	Preceding element can occur one or more times <code>*</code>	gro+ve
?	Preceding element can occur zero or one times <code>*</code>	colou?r
		Alternator, OR operator <code>*</code>   (jpg gif png)
^	Expression after <code>^</code> must be at beginning of line to match	^Hello
\$	Expression before <code>\$</code> must be at end of line to match	Goodbye\$

\	Escape the next character so it is not used as Regex.	image.jpg
\d	Any digit	20\d\d-20-03
\D	Anything NOT a digit	^\D+
\w	Any word character (alphanumeric or underscore)	\w_archive.sql
\W	Anything NOT a word character	\w+\W+\w
\s	White space (space, tab, line break)	\w+\s\w+
\S	Anything NOT white space	\S+\s\S+

There are 3 expressions above that have been marked with an asterisk  .  
These are the **Extended Regular Expression Syntax**.

## Regex Character Classes

Class	Represents
[[:alpha:]]	Alphabetic Characters
[[:digit:]]	Numeric Characters
[[:alnum:]]	Alphanumeric Characters
[[:lower:]]	Lower-case Alphabetic Characters
[[:upper:]]	Upper-case Alphabetic Characters
[[:punct:]]	Punctuation Characters
[[:space:]]	Space Characters (space, tab, new line)
[[:blank:]]	Whitespace Characters
[[:print:]]	Printable Characters (including space)
[[:graph:]]	Printable Characters (NOT including space)
[[:cntrl:]]	Control Characters (non-printing)
[[:xdigit:]]	Hexadecimal Characters (0-9, A-F, a-f)

## Using Regex with Grep

Things to watch out for:

Be careful about mixing up Regex expressions inside Grep, and Unix special characters. So `*` inside grep means something different to its Unix equivalent.

If we use the **Extended Regular Expression Syntax**, we need to add in the `-E` option. Like so:

```
grep -E 'ap+le' fruitbasket.txt
```

If we leave out the `-E`, it won't work.

# TR: Translating Characters

---

TR is short for "translate", and what it does is it copies from the input it receives to output, but with the substitution of selected characters with translation rules that we give it.

```
tr <string to search for> <replacement string>
```

e.g.

```
echo 'a,b,c' | tr ',' '-'
```

To really see how powerful it is, we can use `tr` like this:

```
echo '572089421' | tr '1234567890' 'ABCDEFGHIJ'
```

It takes all the 1's and turns them into A's, etc.

If something is not in our translation string, it simply won't get mapped and will be left alone.

To use character sets with `tr`, use:

```
echo 'This is a secret message' | tr 'A-Za-z' 'N-ZA-Mn-za-m'
```

This will move all the characters 13 places (aka ROT-13)

```
Guvf vf n frperg zrffntr          # Outputs the encrypted message
```

To unencrypt it, use

```
echo 'Guvf vf n frperg zrffntr' | tr 'A-Za-z' 'N-ZA-Mn-za-m'
```

Because the English language has 26 characters, moving it 13 places twice

brings you back to where you started from.

What `tr` is NOT for:

```
echo 'good day' | tr 'day' 'night'
goon nig          # output
```

What it has done here is that it has searched the string, then swapped any 'd' for an 'n', then swapped any 'a' for an 'i', then swapped any 'y' for a 'g'.

If the replacement set is smaller than the search set, then the last item in the replacement set will repeat. If the replacement set is bigger than the search set, then the extra characters will simply never get reached. E.g.

```
echo 'abcdefg123455' | tr 'bd1-4' 'x'
axcxefgxxxx55      # output

echo 'abcdefg123455' | tr 'bd1-4' 'xz'
axczefgzxxx55      # output
```

So here you see that `b` is swapped for `x`, but everything else `d1-4` is swapped for `z`.

To translate upper case to lower case:

```
echo 'KeViN' | tr 'A-Z' 'a-z'
```

we can also use Regex to achieve the same goal. e.g.

```
echo 'KeViN' | tr '[:upper:]' '[:lower:]'
```

## Deleting and Squeezing Characters

By using certain options, `tr` can also delete or filter out certain characters, and to de-dupe repeating characters (squeezing).

Option	Description
-d	Delete characters in listed set
-s	Squeeze repeats in listed set
-c	Use complimentary set
-dc	Delete characters not in listed set
-sc	Squeeze characters not in listed set

E.g.

```
echo "abcd1234aaa1111" | tr -d [:digit:] # abcdaaa
echo "abcd1234aaa1111" | tr -dc [:digit:] # 12341111
```

Using the `-dc` option will strip out any character that is not a digit, including white space characters.

```
echo "abcd1234aaa1111" | tr -s [:digit:] # abcd1234aaa1
echo "abcd1234aaa1111" | tr -sc [:digit:] # abcd1234a1111
```

The first expression above says squeeze (remove duplicates) anything that is a digit. The second expression says squeeze anything that is NOT a digit.

You can also combine the options, like so:

```
echo "abcd1234aaa1111" | tr -ds [:digit:] [:alpha:] # abcd
echo "abcd1234aaa1111" | tr -dsc [:digit:] [:digit:] # 12341
```

Because you have 2 options together, the first argument is going to be the thing you want to delete. The second argument is going to be the thing you want to squeeze.

In the second expression, it's important to note that the `-c` option will only apply to the delete, NOT the squeeze. So it will delete anything that is not a digit, and

then squeeze the digits.

## Windows -> Unix Conversion

Return surplus carriage return and end of file characters:

```
tr -d '\015\032' < windows_file > unix_file
```

## Using SED: Stream Editor

---

SED is a Unix program (short for Stream Editor). SED modifies a stream of input according to a list of commands before passing it on to the output. It's very similar to the way that TR works, but with SED, we can modify the stream of input in many more ways.

SED is a complex tool that offers several modes of working and many features.

It's most use is substitution.

```
sed 's/a/b'
```

**s** is for substitution **a** is the search string **b** is the replacement string

So we're going to search for whatever is in **a**, and replace it with whatever is in **b**.

The difference between this and **TR** is that **TR** uses 2 different arguments for search string and replacement string. Here, they're all inside 1 argument, and they're inside those delimiters.

SED is really based on this one expression.

This is different to TR because TR was translating each thing. This is more like the traditional find and replace you get in a regular word processor.

```
echo 'leftward' | sed 's/left/right/'
```



will return `rightward` .

SED doesn't do a global search by default, so if you enter

```
echo 'leftward and left' | sed 's/left/right/'
```

it will return `rightward and left` .

This is different to GREP, because when we were GREPing for something, it was finding all occurrences of the search string. Remember, the `G` in GREP stands for GLOBAL, so to do the same thing in SED, we need a `g` modifier:

```
echo 'leftward and left' | sed 's/left/right/g'
```

Regarding the delimiters, SED looks at whatever comes after the first `s` and uses that as a delimiter. This is good because sometimes the character you wanted to use as a delimiter is actually part of the search string or replacement string. So then you'd need to change your delimiter. E.g.

```
echo 'Mac OS X/Unix is really cool!' | sed 's|Mac OS X/Unix|Ben|'
```

You could also use:

```
echo 'Mac OS X/Unix is really cool!' | sed 's:Mac OS X/Unix:Ben:'
```

If you wanted to use the backslash in SED, you could, but you would need to escape all instances of the backslash in the search and replacement string. Switching to a different delimiter is an easier and cleaner technique.

## Working With Files

SED takes a second argument which is a filename.

When using input from a file with TR, we needed to use the `<` character, but with SED you can just put the filename in without it. E.g.

```
sed 's/apple/banana/' fruitbox.txt
```

To redirect the output into a different file:

```
sed 's/apple/banana/' fruitbox.txt > bananabox.txt
```

When working with files, each line gets treated as a stream, so if you don't use the `g` modifier, it will only change the first instance of the search string on each line.

So, if you have a list:

```
apple
mango
banana
strawberry
apple
```

It will change it to:

```
banana
mango
banana
strawberry
banana
```

But if you have more than one instance of the search string per line, you'll need to use the `g` modifier to make sure they all change.

You can have multiple expressions on the same line, you just need to delimit them with the `-e` option.

```
echo 'poop shoot' | sed -e 's/poop/crap/' -e 's/shoot/pipe/'
```

It's easy to remember the `-e` option, because it stands for `edit`.

If you have a lot of SED commands you want to run on a single input stream, you can put all these edit commands into a file, and you can use the `-f` option to

run them all at once

## Using Regex with SED

SED is really similar to GREP. In fact, the only serious difference is that SED adds substitution.

Anything you can find with GREP, you can change with SED.

Like with GREP, there is an issue with basic vs. extended expressions. This will work:

```
echo 'who needs chocolate' | sed 's/[ao]/_/g'
```

But this won't:

```
echo 'who needs chocolate' | sed 's/[ao]+/_/g'
```

To use extended expressions, you need to use the `-E` option, like so:

```
echo 'who needs chocolate' | sed -E 's/[ao]+/_/g'
```

SED doesn't understand the shortcut for tab `\t`. In BASH, you need to hit `Control v` and then the character.

This trick works for other characters as well.

## Back References

Back references are part of regular expressions and SED makes good use of them.

A backreference is an regular expression that meets a certain criteria that you can reference later on.

```
echo "paydate" | sed -E 's/(...)date/\1time/'
```

So here, we're looking for anything that is `...time`. We can then reference that `...` as a variable with `\1`. If we have several back references, the first one will be `\1`, the second will be `\2`, etc..

As above, it won't work. Either we need to escape the brackets, or use the `-E` option to enable extended expressions.

## Using CUT

---

Cut allows you to cut out selected portions of each line of a file. While we could use a SED command to do the same thing, CUT is a much simpler way of doing it.

CUT can cut 3 things: characters, bytes, or fields, and we're always going to need to pick one of those. We'll only do characters and fields here.

```
cut -c 2-10 filename.txt
```

`-c` if for characters, `-b` is for bytes, and `-f` is for fields.

What this will do is that it will return characters in columns 2-10 on each line of the file, excluding everything else. In an `ls -lah` command, this will be the permissions.

If you want to cut multiple pieces from a line, use:

```
cut -c 2-10,13-22,45- filename.txt
```

This will grab columns 2-10, 13-22, and 45 until the end of the line. That is what the dash `-` without an end value represents. We could also use the open dash at the beginning to say, everything from the beginning of the line until column 10, like so `-10`.

We could even pipe `ls -lah` through this, like so:

```
ls -lah | cut -c 2-10,13-22,45-
```

## Using fields with CUT

We need to have a tab delimited file. CUT will use the tabs to figure out where the columns are.

So if you had a tab delimited file called `tabbed_stuff.txt` and you ran it in CUT like so,

```
cut -f 1,3 tabbed_stuff.tsv
```

It would return columns 1 and 3.

If you want to use something else as a delimiter, e.g. a CSV, then you'll need to use the `-d` option.

```
cut -f 1,3 -d "," comma_file.csv
```

There's also a `-s` options that does nothing to lines that don't have delimiters.

## Using DIFF

---

DIFF compares 2 file and reports back on what the differences between them are.

```
diff file_1.txt file_2.txt
```

DIFF then spits out a report of what it found, including a code and an extract of the difference.

If there is a `d` in the reporting code, then it means it detected a deletion.

If there is a `c` in the reporting code, then it means it detected a change.

If there is a `a` in the reporting code, then it means it detected an append.

e.g.

```
2d1
< yo mama is so ugly
```

The numbers on either side of the `d` tell us the line numbers where this would occur in each of the 2 files. The number on the left `2` is from the file on the left: `file_1.txt`, and the number on the right `1` is from the file on the right: `file_2.txt`.

The `<` character before the excerpt lets us know that excerpt comes from the left file, i.e. `file_1.txt`.

With deletions and appends, you'll only get 1 line of excerpt, but with change you'll get 2.

## Diff Options

Option	Description
<code>-i</code>	Case insensitive
<code>-b</code>	Ignore changes to blank characters
<code>-w</code>	Ignore all whitespace
<code>-B</code>	Ignore blank lines
<code>-r</code>	Recursively compare directories
<code>-s</code>	Show identical files

`-r` will allow you to compare 2 directories. It will look for files in each directory that have the same filename and then see if there are differences between them. If the files are identical, it will ignore them, unless you use the `-s` option.

## Controlling Diff's Output Format

Option	Description
<code>-c</code>	Copied context
<code>-u</code>	Unified context
<code>-y</code>	Side-by-side
<code>-q</code>	Only whether the files differ (returns yes or no)

The `-c` option gives you the full file for each, and then puts in a `-` where something was deleted, a `!` when something changes, and a `+` where something was added.

`-y` does the same thing, but side-by-side.

`-u` really smashes everything together and is the GIT format.

If you want to output results from Diff as a file, suffix the file with `.diff`

To only get a summary of the changes, use:

```
diff file_1.txt file_2.txt | diffstat
```

A change inside a file using diffstat will be reported as an insertion and a deletion.

Diff works best on 2 files that are similar. Using it on files that differ greatly is pretty much useless.

## Using XARGS

---

We use Xargs to pass argument lists to commands. XARGS is short for "execute as arguments". It parses an input stream into items and then it loops through each item in that list and passes it to a command.

If you want to pass a file to `wc`, you could try

```
echo "myfile.txt" | wc
```

But this wouldn't work well. It would only count the words in the string given - i.e.

```
myfile.txt .
```

To actually pass `wc` the contents of the file, we'd need to use:

```
echo "myfile.txt" | xargs wc
```

The man pages define what comes after `xargs` as its utility command.

If we wanted to see the command before it ran, we could use the `-t` option, like so:

```
echo "myfile.txt" | xargs -t wc
```

To summarize, XARGS passes its input as an argument to a utility command.

But the power of XARGS is looping through a list of arguments.

To run multiple files through `wc`, we could do:

```
echo "myfile.txt yourfile.csv" | xargs -t wc
```

And that would work nicely, but sometimes you don't want XARGS to run the utility command with ALL of the arguments, you want it to loop. To do this, we need to use the `-n` option to limit the number of arguments it will pass through.

```
echo "myfile.txt yourfile.csv" | xargs -t -n1 wc
```

This will only take 1 argument, and then pass it to `wc`, and then it will LOOP, and then pass another argument to `wc`.

So in the above example, it would run `wc` twice.

To limit the number of lines from a file returned to us, we'd use



```
head somefile.txt | xargs -L 2
```

We can also use a placeholder, similar to a back reference in SED.

```
cat fruitbox.txt | xargs -I {} echo "yo mama is a: {}"
```

That will echo out all of the results in the `{}`.

So we can take that argument and place it in a specific place. You can use any placeholder, e.g.

```
cat fruitbox.txt | xargs -I :FRUIT: echo "yo mama is a: :FRUIT:  
:"
```

XARGS will try to split things up by spaces or control characters. This can cause problems because if one of your arguments has a space in it, XARGS might interpret this as a different argument.

The way to solve this problem is to use the `-0` option to tell XARGS that it should only split things by NULL characters as separators instead of spaces and new lines. The main reason you'd use the `-0` option is

```
ls ~/Library/ | grep 'A.*' | xargs -0 -n1
```

## Ways People Use XARGS

One technique is using a `file manifest`, which is a text file containing file names you've chosen and placed in a particular order, separated by line breaks.

i.e.

```
myfile1.txt  
myfile2.txt  
myfile3.txt
```

If the filename is `file_manifest.txt`,

```
cat file_manifest.txt | xargs cat | less
```

This will output all of the file names, pipe them as arguments into CAT, and then pipe all of that output into LESS so that it paginates, because there'll be a lot of output.

```
cat fruitbox.txt | sort | uniq | xargs -I {} mkdir -p ~/Desktop/fruitbox/{} 
```

This would create a directory for each piece of fruit in fruitbox.txt, with it being named as that piece of fruit.

Imagine that we have 50 workers, and we need to create a directory for every worker.

To kill processes, based on id, in an automated way:

```
ps aux | grep 'badstuff' | cut -c 11-15 | xargs kill -9
```

XARGS works really well with GREP and FIND.

`find` is one of the places that you'll need to use the `-0` option, but `find` also has its own option called `-print0`, which tells `find` to use the NULL character to separate results. This is great, because it acts like a handshake between `find` and `xargs`, since `find` delimits results with a NULL character, and `xargs` loops through them, delimiting them by that NULL character.

To automate the `chmod` of certain files based on a find, use:

```
find test1/ -type f -print0 | xargs -0 chmod 755
```

To automate the creation of backup files, use:

```
find . -name "*fruit.txt" -print0 | xargs -0 -I [] cp [] ~/Desktop/{}.backup
```

To then find and remove those backup files:

```
find ~/Desktop/ -name "*.backup" -print0 | xargs -p -0 rm
```

We can't use the interactive option `-i` with `rm` because `xargs` doesn't allow it. Instead, we need to use the `-p` option which stands for "prompt". This will prompt you before it carries out an action.

Caution: When you perform actions like copy or remove, you don't want to delve too deeply. A great way of avoiding this is by setting the find depth to only the current directory - e.g.

```
find ~/Desktop/ -name "*.backup" -depth 1 -print0 | xargs -p -0 rm
```

To go 2 folders deep, set the depth to 2.

The above statement will only ask you once if you want to delete ALL of them. If you want Unix to ask you for each file, use:

```
find ~/Desktop/ -name "*.backup" -depth 1 -print0 | xargs -p -0 -n1 rm
```

You can also use `find` and `grep` together with `xargs`.

E.g. Imagine you're looking through a directory full of invoices and want to find invoices that have the work "plumbing" in them:

```
find . -name "*.invoices*" -print0 | xargs -0 | grep -li 'plumbing'
```

E.g. 2. Imagine that you're a web developer (haha, I am!) and you want to look through the directory of a theme and find every HTML file that contains the `<h1>` tag:

```
find ~/theme/ -name '*.html' -print0 | xargs -0 grep -l "<h3>"
```

You should think about using `xargs` when you want to perform a command on many items in a list. That list can be stored in a file, or dynamically created with `grep` `find` or others.

## Mac-Only Techniques

---

### Working with the Finder and Unix

If you're working in the Finder, you can drag and drop a folder into Terminal, and it will copy the path there.

You can use this for any command that takes a path.

To do the reverse and open a finder window at a specific path from the Unix command line, use:

```
open ~/Desktop/Work/Project1
```

You can do the same with a file, and the Finder will open it up using the default program. E.g.

```
open ~/Desktop/Work/Project1/index.html
```

To open the current directory or the parent directory

```
open .  
open ..
```

To open an application, use

```
open -a calculator
```

The `-a` option says "go to the application folder and find an app there with the same name"

To open up a file in a specific application, use:

```
open -a TextEdit monologue.txt
```

A time saving technique is to use aliases, so

```
alias chrome='open -a chrome'
```

That way, all you have to type is `chrome`. Remember, you'd have to put this into the `.bashrc` file.

`open` will also accept standard input `stdin`, such as:

```
ls -lah | open -f
```

The `-f` option with `open` will output the contents instead of printing it to the screen

This is good because it allows us to use the power of Unix to construct exactly what we want and then we can send it to a regular Mac application.

We could even look at the `man` pages in Preview. The `-t` option will say "format this as postscript".

```
man -t bash | open -fa Preview
```

We can send parts of tab-separated files to Excel or Numbers this way:

```
cut -f 2,6 accounts.tsv | open -fa Numbers
```

OPEN will also open URL's for you in your default browser:

```
open http://bengrunfeld.com
```

## Clipboard Integration with Unix

---

Unix doesn't have a clipboard, but we can use the Mac clipboard to store stuff from Unix.

Because we can't actually select anything in Unix, we use standard output - i.e.

```
ls -lah | pbcopy
```

PB stands for Paste Board. If you then go to notepad and hit paste, the

```
ls -lah
```

 command will print to your application.

```
cut -f 2,6 accounts.tsv | pbcopy
```

This will copy to the clipboard.

```
pbcopy < myfile.txt
```

If you then do

```
pbpaste
```

It will print whatever is on the clipboard.

We can also direct this output to a file:

```
pbpaste > clipboard.txt
```

We could create an alias to do a sort of stuff on the clipboard, then copy it again.

```
alias pbsort='pbpaste | sort | pbcopy'
```

## Working with Multiple Clipboards

There are 4 clipboards in Mac OS x:

- The general clipboard (used above)
- Find

- Font
- Ruler

When you do a find in your web browser or some other application, it usually copies to the Find clipboard.

To copy to the different clipboards, use:

```
echo "General" | pbcopy -pboard {general/find/font/ruler}
```

e.g.

```
echo "Find" | pbcopy -pboard find
```

Then you can paste it out with:

```
pbpaste -pboard find
```

## Taking Screenshots from the Command Line

---

To take a regular screenshot, use

```
Command + Shift + 3
```

To take an interactive screenshot which allows you to use crosshairs to choose the portion you want to take a shot of, use

```
Command + Shift + 4
```

To take a screenshot from the command line, use:

```
screencapture <name of file to save to>
```

E.g.

```
screencapture ~/Desktop/shot.png
```

This way, we can specify our own filename and destination

Option	Description
<code>-i</code>	Interactive capture (esc to cancel)
<code>-m</code>	Main monitor only (when you have multiple monitors)
<code>-C</code>	Show cursor
<code>-t</code>	Which format you'dlike (png, pdf, jpg, tiff)
<code>-T</code>	Delay in seconds
<code>-P</code>	Open file with Preview
<code>-M</code>	Open file in Mail message
<code>-c</code>	Capture to clipboard

E.g.

```
screencapture -mCP -T 3 -t jpg screen_shot.jpg
```

## Mac Shutdown, Reboot And Sleep with Unix

Usually we do this via the user interface on a Mac, but if we do it via the Unix command line, we can schedule these actions.

The command we use for all 3 is

```
sudo shutdown {h/r/s}
```

`h` stands for halt, meaning shutdown, at a specific time `r` reboot at a specific



time `s` sleep at a specific time

The time format you pass it can be 1 of the following 3 options:

- now
- +minutes
- yymmddhhmm

```
sudo shutdown -h now //shutdown now
sudo shutdown -r now //reboot now
sudo shutdown -h now //sleep now

sudo shutdown -h +45 //shutdown in 45 mins
sudo shutdown -r +45 //reboot in 45 mins
sudo shutdown -h +45 //sleep in 45 mins

sudo shutdown -h 1303200000 //shutdown at midnight on that date
sudo shutdown -r 1303200000 //reboot at midnight on that date
sudo shutdown -h 1303200000 //sleep at midnight on that date
```

Unix does have a `sleep` command, but it works as a timer before the next command is executed. So if we had

```
sleep 5; echo "Good Morning!"
```

Unix would wait 5 seconds, then run the second command.

## Text to Speech

---

You can make your computer say stuff from the command line with `say`

```
say "hello there"
```

You can change the voice (list of all voice is in Settings)

```
say "ben rules" -v Zarvox
```

You can pipe in input

```
echo "ben rocks" | say -v Bells
```

You can even `say` a whole file

```
say -f memoirs.txt
```

You can output a file to an audio file, which is good for visually impaired people.

```
say -f memoirs.txt -o audio_file.aiff
```

Use `say` to tell you when a long process has completed

```
cp -R dir1 dir2; say 'You task has finished'
```

## Using Spotlight from the Command Line

---

Spotlight is a fast find tool based on file metadata. Usually you'll use `find` or `grep`, or you'll just use Spotlight in the UI, but there are times when you'll need Unix to have access to the meta data that Spotlight keeps.

```
mdfind "thing"
```

The `md` stands for meta data. This is the same as going into Spotlight in the UI and typing "thing"

The meta data has much more information contained in it that we are usually used to working with.

To hone down the search, use this, otherwise it will search your entire hard drive.

```
mdfind -onlyin <path> "search term"
```

To use a negative: This will make sure that anything with "day" doesn't come up in the results.

```
mdfind -onlyin <path> "summer night -day"
```

The docs recommend using the `-interpret` option, which forces the command to act EXACTLY as the regular UI Spotlight would.

To ensure that the result is in the file name

```
mdfind -onlyin <path> -name "filename"
```

To tell `mdfind` to end every line with a NULL character so that we can use it with `xargs`:

```
mdfind -onlyin <path> -name "filename" -9 | xargs -9 | open
```

## Meta Data Attributes with Spotlight

```
mdls filename.txt
```

`mdls` stands for Meta Data Listing. This will show us all the metadata for the file. We can use that info then. e.g. search for that info with Spotlight. E.g.

```
mdfind -onlyin ~/Desktop/Studies 'kMDItemDisplayName == "*git*"'
```

When you start using time, can only use single quotes.

```
mdfind -onlyin ~/Desktop/Studies 'kMDItemDateAdded <= $time.to day'
```

Instead of using `today`, you can also have `yesterday`, `this_month`, `this_year`, `now`.

With `now`, you can provide a time relative to it in seconds. E.g.

```
mdfind -onlyin ~/Desktop/Studies 'kMDItemDateAdded <= $time.now(-36000)'
```

We can also use this relative time option with `today` .

```
mdfind -onlyin ~/Desktop/Studies 'kMDItemDateAdded <= $time.today(-2)'
```

This means 2 days ago. You can do the same thing for `yesterday` , `this_month` , `this_year` , etc

You can also use an absolute time format with `iso` .

```
mdfind -onlyin ~/Desktop/Studies 'kMDItemDateAdded <= $time.iso(2010-01-01 12:35:00 -0930)'
```

## Applescript with Unix

---

We can do some really cool things with Applescript. The way we're going to call applescripts is the use the Mac-only command

```
osascript {filename}|{script}
```

`osa` stands for open scripting architecture. After `osascript` , we can either provide a path to a filename, or an individual script. E.g.

```
osascript -e 'set volume output muted true'
osascript -e 'set volume output muted false'
```

We can also use Applescript to talk to different applications, like the Finder.

```
osascript -e 'tell application "Finder" to display dialog "Hello"'
```

If you're going to create a file, suffix it with `.scpt` .

There is also a command called `osacompile` which compiles scripts.

## System configurations: Viewing and Setting

---

There are hundreds of configuration options on a Mac. You can see some of them, but you can access many more via Unix.

Some changes require a logout and login. Others require a total shutdown.

The different configurations are stored both in `Ben/Library` , and some are stored in `Macintosh HD/Library` .

The settings in `Macintosh HD/Library` apply to every user on the system. The settings in `Ben/Library` apply to just me.

Inside `Library` for each of them, there's also a `Preferences` folder. That's where the settings are typically stored, although sometimes they exist in other places.

Usually the format of the files is

```
com.apple.<application name>.plist
```

`plist` lets you know that it's a preferences list. If you open a `plist` file, it will open the Mac preferences editor, where you can change some of the values.

To change these values in Unix, we'll use the `defaults` command.

`defaults` takes care of whether the preference is a user preference or a system preference. It goes and finds it in the right place and changes it there.

```
defaults read <domain> <key>
defaults write <domain> <key> <value>
```

Domain is typically in the format: com.companyname.appname. E.g.

```
defaults read com.apple.finder CopyProgressWindowLocation
defaults write com.apple.finder CopyProgressWindowLocation "16
8, 270"
```

Sometimes a preference file won't exist in the Library. Defaults will then go and create the necessary file.

The best place to learn about these is on the internet.

## Examples of System Configurations

To show dot files  in Finder

```
defaults write com.apple.finder AppleShowAllFiles -bool TRUE
```

To display Unix path in Finder windows

```
defaults write com.apple.finder _FXShowPosixPathInTitle -bool
TRUE
```

To modify the screen capture file type:

```
defaults write com.apple.screencapture type PNG
```

Formats available: PNG, BMP, GIF, JPEG, PDF, PICT, PSD, SGI, TGA, TIFF

To modify the screen capture file save location:

```
defaults write com.apple.screencapture location "/Users/Ben/De
sktop"
```

To change the login screen background:

```
sudo defaults write com.apple.loginwindow DesktopPicture "/Lib
rary/Desktop Pictures/Aqua Blue.jpg"
```

We need `sudo` because this is a system level preference.