Introduction to Command-line

A brief introduction to OSX command-line wizardry with extensive examples. For purposes of this introduction **command line**, **shell**, **terminal** and **bash** are all the same thing. There are specific, technical meanings for each of these words, but right now you shouldn't care about that right now.

Starting the terminal

Click the **Applications** folder in your dock, then **Utilities**, then **Terminal**. You'll be faced with something that looks a little like this:

```
$
```

This is where you type your commands. When you see the `$` in the following examples, *don't* type it. It's just there to show you where commands are being entered. Press enter after you type a command to execute it.

Put your mouse/trackpad in a drawer. There are no mice in the command line.

Running your first command

Type the command after the dollar sign and hit enter:

```
$ echo hello
hello
```

`echo` just outputs what you tell it the command line. That's all it does. This may not seem very useful, but it illustrates an important concept. On the command line you type **commands**, like `echo`. These commands take **arguments**, like `hello`.

Where am I?

The command line has a sense of *location*, that is, what folder you are *in* at any given time. This is called your **current working directory**. (**Directory** is just another name for a "folder".) To see what directory you are in right now:

```
$ pwd
/Users/onyxfish
```

`pwd` is short for "print working directory". By default the terminal always starts you in your **home directory**, which is named with your login **username**. This is the same directory which contains your `Documents` folder on OSX, so you are probably familiar with it.

What's here?

To see what files are inside the **current working directory**, use the `ls` command:

```
$ ls
Applications            bin
Confidential            confidential.py
Desktop                 confidential2.tc
Documents               external_lbrs
Downloads               feed.json
Dropbox                 feed.py
Library                 floobits
Movies                  gitconfig
Music                   google_analytics_auth.dat
Pictures                pgadmin.log
Public                  s3cfg
SpiderOak Hive          src
VirtualBox VMs          syria
adt-bundle-mac          tmp
android-sdk-macosx
```

This is all the stuff in my home directory. Yours will look different.

What's in there?

What if you want to see what files are in a different directory?

```
$ ls Documents
Arduino             android-workspace
Aspyr               eagle
GoPro Projects      gopromote.prm
Hero Lab            nicar_expenses.pdf
Klei                nicar_receipts.pdf
Library             orchard bank travel number.txt
Logan.game          pedagogy.txt
Logans.band         piano 1.band
MapBox              test.txt
NACIS notes.txt     tyler-env.txt
Roblox
```

By passing `Docuemnts` as an **argument** to `ls`, it's shown me the files in my `Documents` directory, which is inside my **home directory**.

Some things are hidden

By default, the `ls` will not show you files whose names being with a **period**. These are called **hidden files** are are usually related to configuring programs. However, it's very often useful to see these files, which you can do with the `-a` flag:

```
$ ls -a
.                       .pypirc
```

```
..                      .python-eggs
.CFUserTextEncoding     .qgis
.DS_Store               .qgis2
.MacOSX                 .relay.conf
.NERDTreeBookmarks      .rnd
.Platformer             .rstudio-desktop
.Rhistory               .s3cfg
.Rube Goldberg          .spyder2
.Trash                  .sqlite_history
.Xauthority             .ssh
.android                .subversion
.anyconnect             .teamocil
.bash_history           .tilemill
.bash_profile           .tmux.conf
.build                  .tox
.cache                  .uibtedbn
.clan_auth.dat          .vagrant.d
.clan_secrets.json      .vim
.config                 .viminfo
.cordova                .viminfo.tmp
.cups                   .viminfz.tmp
.distlib                .vimrc
.dropbox                .virtualenvs
.dropbox-master         .wireshark
.eaglerc                .wireshark-etc
.ec2                    .ypp_42
.floorc                 .zcompdump
.fontconfig             .zcompdump-nomad-5.0.2
.gem                    .zprofile
.gitconfig              .zsh-update
.gitsh_history          .zsh_history
.gnome2                 .zshrc
.gnupg                  Applications
.goaccessrc             Confidential
.godot                  Desktop
.haxelib                Documents
.heroku                 Downloads
.hkzftsorc              Dropbox
.hxcpp_config.xml       Library
.hxcpp_config.xml.bak   Movies
.ievms                  Music
.infinit                Pictures
.inkscape-etc           Public
.ipython                SpiderOak Hive
.keybase                VirtualBox VMs
.keybase-installer      adt-bundle-mac
.lesshst                android-sdk-macosx
.lighttable             bin
.local                  confidential.py
.matplotlib             confidential2.tc
.mongorc.js             external_lbrs
.netrc                  feed.json
.ngrok                  feed.py
.node-gyp               floobits
```

```
.npm                    gitconfig
.oh-my-zsh              google_analytics_auth.dat
.pgadmin_histoqueries   pgadmin.log
.pgpass                 s3cfg
.pip                    src
.psql_history           syria
.pylint.d               tmp
```

You're most likely to encounter these files if someone asks you to add something to your `.bash_profile` file in your **home directory**.

Making a path

In order to be effective on the command-line, you need to understand **paths**. A path is a series of **directory** names, separated by `/` and sometimes ending with a filename. A **directory** is just another name for a folder. Let's go back to `pwd`:

```
$ pwd
/Users/onyxfish
```

So `/Users/onyxfish` is the **path** to my **home directory**.

What if wanted the path to the `src` directory that is inside my home directory. Then it would be:

```
/Users/onyxfish/src
```

And if we wanted the `tshirt` project directory inside that `src` directory?

```
/Users/onyxfish/src/tshirt
```

And if we wanted the `README.md` file inside the tshirt project?

```
/Users/onyxfish/src/tshirt/README.md
```

Going there

Now that you can create a path, let's go there!

```
$ cd /Users/onyxfish/Documents
```

`cd` is short for "change directory", and it changes your **current working directory**. Now if you type `ls` you will see the files in your `Documents` folder:

```
$ ls
Arduino             android-workspace
Aspyr               eagle
GoPro Projects      gopromote.prm
Hero Lab            nicar_expenses.pdf
Klei                nicar_receipts.pdf
Library             orchard bank travel number.txt
Logan.game          pedagogy.txt
Logans.band         piano 1.band
MapBox              test.txt
NACIS notes.txt     tyler-env.txt
Roblox
```

Making paths relative

You don't always have to type the full path to a file. You can refer to a file by it's path **relative** to your current directory. For example, from my **home directory** these two paths are equivalent:

```
/Users/onyxfish/src/tshirt/README.md
src/tshirt/README.md
```

The first path is an **absolute** path, meaning that it starts at the **root** of my hard drive, as designated by the leading `/`.

The second path is a **relative** path, meaning that it starts at your current working directory, the same directory that was listed when we ran `pwd`.

Going home

The command line includes a special symbol that always refers to your **home directory**. This is the `~`. Whenever you type this symbol, it is the same as typing the path to your home directory, so when you need to get back to your home directory, simply type:

```
$ cd ~
```

You can also use the `~` to create paths. All of these paths are equivalent:

```
/Users/onyxfish/src/tshirt/README.md
src/tshirt/README.md
~/src/tshirt/README.md
```

What's up there?

Relative paths and the `~` are not the only ways to save time when constructing paths. It's also important to know about `.` and, especially `..`

`.` always refers to the directory you are in. The uses for this are not important now. `..` always refers to the directory that *contains* the one your are in, that is, the directory *above* your **current working directory**.

Let's say that I'm working on the `tshirt` project and I want to work on the `commencement` project. My directories look like this:

```
~
    src
        tshirt
        commencement
```

That is, my **home directory** contains the `src` directory, which contains the `tshirt` and `commencement` directories. I'm currently working in the `tshirt` diretory:

```
$ pwd
/Users/onyxfish/src/tshirt
```

To get to the commencement directory I could type the **absolute path**:

```
$ cd /Users/onyxfish/src/commencement
```

But this is long-winded and I have to remember the entire path. Instead, I can construct a **relative path** using the `..` syntax:

```
$ pwd
/Users/onyxfish/src/tshirt
$ cd ..
$ pwd
/Users/onyxfish/src
$ cd commencement
$ pwd
/Users/onyxfish/src/commencement
```

Here you can see we've used the `..` to move the directory above tshirt and then used a relative path to `commencement` to move into that directory. We can simplify this even further:

```
$ pwd
/Users/onyxfish/src/tshirt
$ cd ../commencement
$ pwd
/Users/onyxfish/src/commencement
```

In this example we've used the `..` to construct a relative path directly to the `commencement` directory. Note that I never had to remember the names of the directories containing these files to get from one to the other.

Tab completion saves the day

Even knowing about `~` and `..` you are still going to be typing a lot of folder names. Fortunantly, your **command line** supports **tab completion** for both **commands** and **directories**. To see this in action, type this, but instead of hitting enter after the second command, hit the Tab key where indicated in the second command. (Don't type `[TAB]`.)

```
$ cd ~
$ cd Docu[TAB]
```

Tab completion will auto-complete `Docu` to `Documents/`. **Remember this, you will use it constantly.**

Cancel what you're doing

Now if you followed directions you have `$ cd Documents/` sitting in your command line. Let's say we don't actually want to run this command. We've changed our mind. To clear this command and reset back to and empty terminal hold the `control` key and press `c`. You'll frequently see this abbreviated `CTRL+C`.

```
$ cd Documents/[CTRL+C]
$
```

It's useful to imagine that the `c` stands for cancel. As you'll see later on, `CTRL+C` is often used to exit command line programs, in addition to clearing what things you've typed, but then changed your mind about.

Create a file

You can create an empty file using the `touch` command. You won't use this often, but it's very helpful for what we're going to do next.

```
$ touch test.txt
```

Now if you run `ls` you'll see `test.txt` in the list of files.

Delete a file

To delete a file use the `rm` command and provide it's filename as an **argument**.

```
$ rm test.txt
```

Now if you run `ls` you'll see that `test.txt` is no more.

Take not that this is a little different than right-clicking a file in Finder and selecting "Move to Trash" because there is no Trash Can on the command line. Once you delete it, it's simply gone.

Create a directory

Directories can be created using the `mkdir` command. This is equivalent to right-clicking in Finder and selecting "New Folder":

```
$ mkdir test
```

Delete a directory

Directories are also deleted with `rm`, however, you have to do something special to delete a directory:

```
$ rm -r test
```

`-r` is an example of a **flag**. A flag is a sort of switch that you provide to a **command**. In this case the `-r` flag is short for "recursive", which tells `rm` to delete both `test` and *everything inside it*.

To summarize:

- `rm` is a **command**.
- `-r` is a **flag**.
- `test` is an **argument**.

Flags *usually*, but not always, come before **arguments**.

Be very careful with the `-r` flag. It's a really easy way to delete a lot of things.

Delete just those files

What if I wanted to delete all the JPEGs in directory, but nothing else? First, let's create some dummy JPEG files:

```
$ touch test1.jpg
$ touch test2.jpg
$ touch test3.jpg
```

To delete these three files we can use the `*` symbol, known as a **glob** or **wildcard**. The `*` means "match anything that matches both before and after". Let's look at some examples:

```
$ rm *.jpg
```

This will delete all files in the **current working directory** that end in `.jpg`.

```
$ rm test*
```

This will delete all files in the current working directory that begin with `test`.

```
$ rm test*.jpg
```

This will delete all files in the current working directory that begin with `test` and end with `.jpg`. This is the safest of these commands as it is the least likely to accidentally match something else.

Be careful when using globs that you don't accidentally delete other things.

Finding the file you need

Finding files in the command line is an obvious thing you might need to do, and yet even some veteran terminal users can't remember how to do it off the top of their had. The `find` **command** is a good example of an exception to the rules of the command line. In particular, it uses **flags** and **arguments** in a different way than the other commands we've looked at. Let's create a file and find it:

```
$ touch test.txt
$ find . -name test.txt
./test.txt
```

In this case:

- `find` is the **command**.
- `.` is an **argument**, indicating what directory to search in. As you'll recall, `.` always refers to the current directory. This will also search all directories within the current one, the directories within those directories and so on.
- `-name "test.txt"` is a **flag** which takes a **value**, in this case, the name of the file you are looking for.

You can also combine `find` with the `*` to find files that match multiple files, for example:

```
$ touch test1.txt
$ touch test2.txt
$ find . -name test*.txt
./test1.txt
./test2.txt
```

Learning more about a command

In addition to searching for filenames you can use `find` to search by creation date, update date, file size and more. You can learn more about many command line **commands** by using the `man` command:

```
$ man find
```

This will open the builtin documentation for the command, which will explain all the **flags** and **arguments** it accepts. Not all commands have builtin documentation, but most do. Within the documentation you can use your **arrow keys** to navigate by line, `f` (*forward*) and `b` (*forward*) to navigate by pages and `q` to quit.

Download a file

One of the most common things you may need to do is grab a file from the internet so that you can work with it locally. This might be an image, the output of an API or who knows what else. For experimenting, let's download the text of James Joyce's Ulysses from Project Gutenberg.

```
$ curl -o ulysses.txt http://www.gutenberg.org/cache/epub/4300/pg4300.txt
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1536k  100 1536k    0     0   736k      0  0:00:02  0:00:02 --:--:--  737k
```

The `-o` **flag** allows us to specify the an output filename for the download.

Browsing output

Of all the data formats you might work with–spreadsheets, databases, images–the command line has a particularly robust set of tools for working with **text**. Text in this case means **plain text**, not Word documents, RTF or any other format.

Let's us the the `cat` **command** to print the contents of our text file:

```
$ cat ulysses.txt
...
[lots of text here]
...
This Web site includes information about Project Gutenberg-tm,
including how to make donations to the Project Gutenberg Literary
Archive Foundation, how to help produce our new eBooks, and how to
subscribe to our email newsletter to hear about new eBooks.
```

What happened here? All the text of Ulysses just scrolled past on your screen. The `cat` command is like `echo` except instead of printing whatever **argument** you give it, it prints the contents of a file. It doesn't care how long that file is. To browse a long file, we should instead use `less`:

```
$ less ulysses.txt
<U+FEFF>The Project Gutenberg EBook of Ulysses, by James Joyce

This eBook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever.  You may copy it, give it away or
re-use it under the terms of the Project Gutenberg License included
```

```
with this eBook or online at www.gutenberg.org
...
[more text here]
...
ulysses.txt
```

Within `less` the commands to navigate are the same as within `man`. Use the **arrow keys** to navigate by lines, `f` (*forward*) and `b` (*backward*) to navigate by pages and `q` to quit.

Note: there are many types of files which although not strictly **text** are never-the-less text-based formats, for example: CSV, YAML and JSON. In general the builtin command line tools are useful for processing these kinds of files as well.

Search for text within a file

`more` is useful for skimming a text file, but what if you're looking for specific text? That's where `grep` comes in:

```
$ grep stately ulysses.txt
stately figure entered between the newsboards of the _Weekly Freeman
Mrs M'Guinness, stately, silverhaired, bowed to Father Conmee from the
```

`grep` takes two **arguments**: the text to search for and the name of the file to search within. It searches, line-by-line, and outputs every line where that text appears.

You can also use the `-c` flag to output a count of the number of occurences:

```
$ grep -c lovely ulysses.txt
64
```

Note: you can't use `*` in your text searches. **Glob** syntax does not work in this case. You must instead use regular expressions, which are beyond the scope of this tutorial. You **can** use the `*` to select multiple files to search!

Search for more

In the previous examples, we we're searching for only a single word. If we tried to search for multiple words our output would be confusing:

```
$ grep -c lovely socks ulysses.txt
grep: socks: No such file or directory
ulysses.txt:64
```

`grep` treats every **argument** after the first one as a file to search. In order to search for a phrase, we need to **quote** the search **argument**:

```
$ grep "lovely socks" ulysses.txt
the tie he wore, his lovely socks and turnedup trousers. He wore a pair
```

It's good to get in the habit of quoting text **arguments**. For example, early on you ran:

```
$ echo hello
```

Though this works, it would be better to quote the argument:

```
$ echo "hello world!"
```

Write output to a file

You now know enough for me to introduce you to some of the most powerful concepts in the command line: **standard in** and **standard out**. It's helpful to think of these as channels through which data can travel into and out of a program. In the very simplest case, **standard out** is simply what you see printed to the command line after you run a command.

```
$ grep stately ulysses.txt
stately figure entered between the newsboards of the _Weekly Freeman
Mrs M'Guinness, stately, silverhaired, bowed to Father Conmee from the
```

In this example, lines 2 and 3 are printed to **standard out**. By default, anything printed to standard out appears on the command line after your command. But what if this isn't what we want? What if you wanted to save your results into a file? You need **redirection**:

```
$ grep stately ulysses.txt > search.txt
```

The `>` is short for "redirect standard out to a file."

```
$ cat search.txt
stately figure entered between the newsboards of the _Weekly Freeman
Mrs M'Guinness, stately, silverhaired, bowed to Father Conmee from the
```

Here we see that the results of our search are now saved in the file named `search.txt`. If we we're to run this command again it would overwrite this file with the same results. What if we wanted to add *more* results to this file?

```
$ grep lovely ulysses.txt >> search.txt
```

The `>>` is short for "redirect standard out to a file, **appending** to it's contents." If you `cat search.txt` now, you'll see it contains our search results for both "stately" and "lovely".

Fun with pipes

Standard out is useful because you can redirect it to files, but it's got another, even more useful function: **redirecting** to **standard in**. As I said before, standard in is a sort of channel through which you can pass data into your **command**. You can't type into this channel, but you can send data into it using the `|`.

```
$ cat ulysses.txt | grep stately
stately figure entered between the newsboards of the _Weekly Freeman
Mrs M'Guinness, stately, silverhaired, bowed to Father Conmee from the
```

`cat` reads a file and outputs its contents onto **standard out**. As you'll recall when we did this before it output the entire contents of the file, so we used `less` instead. In this case, we output the entire text of the file, but use the `|` symbol, which is short for "redirect standard out into standard in". `grep`, in addition to accepting a filename, can also accept data on **standard in**. Thus these two commands are functionally identical:

```
$ grep stately ulysses.txt
$ cat ulysses.txt | grep stately
```

Because of the symbol, we call this **piping**, but it's just a special kind of **redirection**. Let's say that we want to search for lines that contain the word "mother" and the word "father", but not necessarily together:

```
$ cat ulysses.txt | grep father | grep mother
     My mother's a jew, my father's a bird.
are the dispossessed son: I am the murdered father: your mother is the
excellent idea of affording the poor fatherless and motherless children
a bloody fool to it. Handed him the father and mother of a beating. See
he knows if he's a father or a mother.
from father to, mother to daughter, I mean. Bred in the bone. Milly for
father and thy mother that had the best hand to a rolypoly or a hasty
father and mother of a bating. _(With a tear in his eye)_ All insanity.
Kingsbridge station with his father and mother I was in mourning thats
```

In this example we've **piped** twice, first into a `grep father` and then into a `grep mother` to further filter the results. Want to save the final results to a file? Just **redirect** the output again:

```
$ cat ulysses.txt | grep father | grep mother > search.txt
```

Play it again

As you get used to using pipes you may find that your commands are getting very long. At some point the overhead of making a typo gets very frustrating. You don't want to have to retype very long commands. Fortunantely, you can always go back to a command you executed before by pressing the **up arrow** at the command line. If you press it more than once, you can cycle back through the last commands you've executed. Just hit the `enter` key once you've found the one you want.

Sometimes the command you want is one you just entered, but one you entered an hour ago. In this case using the **up arrow** to find it may be very tedious. Instead, hold the `control` key and push `r`. This will put you into "reverse search mode". Simply type in characters that appeared in your original command and terminal will find it. If you've run several commands with similar names you may have to type more characters to uniquely identify it. Once you've found the command you want, press `enter` to run it or the **right arrow** to put it into the command line but not run it.

Editing text

nano (+vim)

Becoming root

sudo

Do it all the time

aliases, scripts, environment vars

Go crazy

python, csvkit, sqlite, xargs?