# Playing with text on the command line

The command line (also known as the command line interface, or CLI, or sometimes the terminal), is a plain text-based interface for executing commands on a computer. If you've ever seen a movie about hackers from the 1980s, like *WarGames*, where they stare at a prompt on a black screen and type in commands one at a time, it's basically that.

You have a prompt, and you can type in a command and hit 'Enter' to execute it. An example command would be:

```
touch newfile.txt
```

This command will create a file called `newfile.txt`.

## How to access the command line

**Mac OS X:** Go to /Applications/Utilities and click on "Terminal" or search for "Terminal" in Spotlight.

**Desktop Linux:** You can search for the "Terminal" application from the Dash. Let's be honest, though, if you're running Linux, you probably don't need this tutorial.

**Windows:** Windows is a bit of a special case. If you go to the Start Menu and click "Run", and then type "cmd" and hit enter, it will open the Windows version of the command line. Unfortunately, the Windows version of the command line kind of has its own system, so for the purposes of following these examples, you'll want to install Cygwin, which will allow you to mimic a Linux-style command line:

http://www.cygwin.com/

## A little more detail

Commands generally take the format:

```
[name of the command] [option] [option] [option] ...
```

The prompt will also show what directory you're currently sitting in. Whenever you execute a command, you do it from a particular directory. This matters because when you execute a command that involves a filename or a directory name, you can specify it one of two ways:

**Relative Paths** Specifying a file or directory as a relative path means you are specifying where it sits relative to the directory you're in. For example, let's say you're in the `videos` subdirectory of the `files` directory. You'll see this prompt:

```
/files/videos$
```

If you execute a command like `touch newfile.txt`, it will create `newfile.txt` inside the current directory. Relative paths don't start with a slash.

**Absolute Paths** Specifying a file or directory as an absolute path means you are specifying where it sits on the computer in absolute terms, starting from the top level. For example, let's say you're in the `videos` subdirectory of the `files` directory again.

```
/files/videos$
```

If you execute a command like `touch /files/music/newfile.txt`, it will create `newfile.txt` inside a different folder, the `music` subfolder of the `files` folder. *Absolute paths start with a slash.*

If you use an absolute path, the command will do the same thing no matter what directory you execute it from.

So these two commands will have the same result from the `/files/videos` directory:

```
/files/videos$ rm video.mp4
(This will delete the file `video.mp4` from the current directory)

/files/videos$ rm /files/videos/video.mp4
(This will delete `video.mp4` from the /files/videos/ directory, which happens to be the cu
```

The same two commands will not have the same result if you are in a different directory:

```
/files/text$ rm video.mp4
(This will try to delete the file video.mp4 from the 'text' subdirectory instead, because th

/files/text$ rm /files/videos/video.mp4
(This will delete the file from the /files/videos/ directory, even though it isn't the curre
```

Remember:

**Starting a path with a slash** means you want to give the entire path and ignore what directory you're currently in. **Not starting a path with a slash** means you want to give the path starting from the directory you're in.

If you're ever unsure of what directory you're in, you can use the `pwd` (Print Working Directory) command to get the absolute path of the current directory.

```
~$ pwd
/Users/Noah
```

## File Patterns

In most cases when you have to specify a file name or directory name, you can also specify a general **pattern** that might match multiple files. There are lots of ins and outs with this, but the most basic version is using the asterisk (*), which matches anything. It's also known as a wildcard.

```
Delete any file in the current directory
/files$ rm *

Delete any file that ends in '.txt'
/files$ rm *.txt

Delete any file that starts with 'data'
/files$ rm data*
```

## Navigating

The two core commands for navigating what directory the prompt is in are `cd` and `ls`.

`cd` is a command to change the current directory, and must be followed by a directory you want to change to. You can supply an absolute or relative path.

```
This will put you in /files/videos
/files$ cd videos
/files/videos$

This will put you in /videos, and then the vines subdirectory
/files$ cd /videos
/videos$ cd vines
/videos/vines$
```

You can jump multiple levels at once if you want.

```
This will put you in /files/videos/short
/files$ cd videos/short
```

You can use `cd ..` to move up one level to the parent directory.

```
This will put you in /files
/files/videos$ cd ..
```

`ls` will list the files in the current directory. It's helpful for figuring out where you are, what files exist, and what subfolders exist.

```
/photos$ ls
thumbnails  photo1.jpg  photo2.jpg
```

Using `ls -l` will print the list vertically, with lots of other extra information about the file size, permissions, and last modified date:

```
/photos$ ls -l
-rw-rw-r-- 1 noah noah 58133 Oct 22 17:13 photo1.jpg
-rw-rw-r-- 1 noah noah 75640 Oct 22 17:13 photo2.jpg
drwxrwxr-x 2 noah noah 4096  Oct 22 17:13 thumbnails
```

When typing in a directory or file name, you can hit the 'Tab' key to autocomplete if it's possible. For example, in the /photos folder, if you type in:

```
/photos$ cd thu
```

and hit 'Tab,' it will fill in the rest and show you:

```
/photos$ cd thumbnails
```

However, if there is more than possible file/directory that matches what you've typed so far, it won't work. If you type:

```
/photos$ rm pho
```

and hit 'Tab,' nothing will happen because you could be on your way to `photo1.jpg` OR `photo2.jpg`.

## Command Output

The commands we're going to talk about all output their results as text. When you execute the command by hitting 'Enter', it will print out a bunch of output on extra lines below the prompt. For example, `head [file]` will print out the first 10 lines of a file.

```
/files$ head names.txt
Dan Sinker
Erika Owens
Noah Veltman
Annabel Church
Friedrich Lindenberg
Sonya Song
Mike Tigas
Brian Abelson
Manuel Aristaran
Stijn Debrouwere
/files$
```

Notice that after it prints out its output, it goes back to giving you a fresh prompt. Getting the output printed out to you in this fashion is useful if you're just poking around, but often you want to do one of two things: **send the output to a file**, or **send the output to another command as an input**.

### Sending the output to a file

You can send the output to a new file this way:

```
/files$ head names.txt > first10names.txt
```

If first10names.txt doesn't exist, it will be created. If it already exists, it will be overwritten.

You can append the output to the end of an existing file this way:

```
/files$ head names.txt >> allnames.txt
```

This will add the output as 10 new lines at the end of allnames.txt.

**Sending the output to another command as an input**

You can send the output to another command using the pipe symbol (|). The `grep` command searches through some text for matches (more on this later), so you could do this to get the first 10 lines of a file, and then search for "Steve" within those 10 lines:

```
/files$ head names.txt | grep "Steve"
```

This is basically the same as doing this:

```
/files$ head names.txt > temporaryfile.txt
/files$ grep "Steve" temporaryfile.txt
```

But instead of first sending the output to a file and then running the second command on that file, you pipe the output directly from the first command into the second. You can chain as many of these together as you want:

```
/files$ grep "United States" addresses.csv | grep "California" | head
```

This would search the file addresses.csv for lines that contain the phrase "United States", then search the results for lines that contain the word "California", and then print out the first 10 of those matches.

## Grep

The `grep` command will let you search a file (or multiple files) for a phrase. By default, it will print out each line that matches your search.

Print out lines that contain the word "darkwing":

```
/files$ grep "darkwing" famousducks.txt
```

Same as above, but the search is case-insensitive:

```
/files$ grep -i "darkwing" famousducks.txt
```

Find matches for the exact *word* "Donald" in a file - words that contain "Donald," like "McDonald," won't count:

```
grep -w "Donald" famousducks.txt
```

Find matches for “McDuck” in every file in the current directory:

```
grep "McDuck" *
```

Find matches for “McDuck” in every file in the current directory AND every subdirectory, all the way down:

```
grep -r "McDuck" *
```

For each match of “Howard”, print out that line AND the 4 lines after it (5 lines total):

```
grep -A 4 "Howard" famousducks.txt
```

For each match of “Howard”, print out that line AND the 4 lines before it (5 lines total):

```
grep -B 4 "Howard" famousducks.txt
```

For each match of “Howard”, print out that line AND the 4 lines before it AND the 4 lines after it (9 lines total):

```
grep -C 4 "Howard" famousducks.txt
```

Instead of printing out the matching lines themselves, print out the filenames that match your search:

```
grep -l "Daffy" *
```

Just get the number of matches:

```
grep -c "Daffy" *
```

Show line numbers along with the matching lines:

```
grep -n "Daffy" famousducks.txt
```

## Cat

The `cat` command will combine multiple files together. This will print three files in a row, as if they were one file:

```
cat turkey.txt duck.txt chicken.txt
```

Remember that this will just print the output into your terminal. More likely, you want to create a new file that combines them:

```
cat turkey.txt duck.txt chicken.txt > turducken.txt
```

turducken.txt will contain all of the lines in turkey.txt, followed by all of the lines in duck.txt, followed by all of the lines in chicken.txt.

If you want to combine ALL of the files in a directory, you can use a wildcard:

```
cat * > allfilescombined.txt
```

## Head

The `head` command will print out the first 10 lines of a file:

```
/files$ head names.txt
```

You can also specify a different number of lines. This will print out the first 15 lines of a file:

```
/files$ head -n 15 names.txt
```

Or, if you want to print all the file but leave out the LAST 15 lines, you can give a negative number:

```
/files$ head -n -15 names.txt
```

One of the nice uses of head is to quickly peek inside a large text file to see what's in it without having to wait for a text editor to load it. This becomes a big deal when you're talking about a 1 GB file!

## Tail

The `tail` command is the reverse of head. It will print out the last 10 lines of a file:

```
/files$ tail names.txt
```

This will print out the last 15 lines of a file:

```
/files$ tail -n 15 names.txt
```

Or, if you want to print all the file but leave out the FIRST 15 lines, you can add a plus sign:

```
/files$ tail -n +15 names.txt
```

This is helpful if you want to, say, remove a header row from a CSV file:

```
/files$ tail -n +1 names.txt > names-no-header.txt
```

## Miscellaneous

If you just want to print out the entire contents of a file into your terminal, you can use `cat` and not combine it with anything. This is sort of against the whole point of `cat`, but is a handy trick.

```
/files$ cat address.txt
1600 Pennsylvania Avenue
Washington, DC 20500
```

If you want to get serious and open a file in a text editor that comes built in to your terminal, you can try `nano`:

```
/files$ nano address.txt
```

How many lines are in names.txt?

```
/files$ wc -l names.txt
18
```

## Regular expressions

When using something like `grep` to search, you can search for a simple term with only letters, numbers, and spaces. But if you want to search for a pattern, you can use what's called a **regular expression**. Regular expressions use special characters to represent patterns, like "any number," "any letter," "X or Y," "at least three lowercase letters," and so on.

We won't worry about the ins and outs for now, but one useful operator is the period (.). In regular expression-ese, this means "One of any character." So you can search for something like:

```
/files$ grep -i "car.s" dictionary.txt
```

This would match words like `cards`,`carts`,`cares`, and so on. It would also match the middle of the phrase "scar story" (CAR S) because "any character" means ANY character, including a space or a punctuation mark.

One more example:

```
/files$ grep -i ".e.st" dictionary.txt
```

This would match things like `least`,`beast`, and `heist`.

## More than one way to skin a cat

There are often lots of equally legitimate commands or combinations of commands to achieve the same purpose.

Example:

```
/files$ head -n 12 names.txt | tail -n 5
(Print out the first 12 lines, and then print out the last 5 lines of that)

is the same as

/files$ tail -n +7 names.txt | head -n 5
(Print out everything but the first 7 lines, then print the first 5 lines of that)

is pretty much the same as:

/files$ tail -n +7 names.txt > temporaryfile.txt
/files$ head -n 5 temporaryfile.txt
/files$ rm temporaryfile.txt
(Save everything but the first 7 lines to a temporary file, then print the first 5 lines of
```