Startup Engineering
Balaji S. Srinivasan
Stanford University

Class: class.coursera.org/startup-001
Signup: coursera.org/course/startup
Forums: bit.ly/startup-eng-forum

The Linux Command Line

In this lecture we'll introduce you to a large number of new command line tools and show you how to connect them together via pipes.

The Command Line

In general, the best way to understand the command line is to learn by doing. We'll go through a number of useful commands with examples, and you should become reasonably skilled in the process, but it is nevertheless worth looking at a few books to consult as references. Here are some particularly good ones:

- The Command Line Crash Course is Zed Shaw's free online tutorial introduction to the command line. Much shorter than the others listed here and recommended.
- Sobell's Linux Book: If you only get one book, get Sobell's. It is probably the single best overview, and much of the content is applicable to both Linux and OS X[1] systems.
- Unix Power Tools: While this came out in 2002, much of it is still relevant to the present day. New options have been added to commands but old ones are very rarely deprecated.
- Unix/Linux Sysadmin Handbook: Somewhat different focus than Sobell. Organized by purpose (e.g. shutdown), and a good starter guide for anyone who needs to administer machines. Will be handy if you want to do anything nontrivial with EC2.

With those as references, let's dive right in. As always, the commands below should be executed after `ssh`'ing into your EC2 virtual machine[1]. If your machine is messed up for some reason, feel free to terminate the instance in the AWS EC2 Dashboard and boot up a new one. Right now we are doing all these commands in a "vanilla" environment without much in the way of user configuration; as we progress we will set up quite a lot.

The three streams: STDIN, STDOUT, STDERR

Most `bash` commands accept input as a single stream of bytes called `STDIN` or standard input and yields two output streams of bytes: `STDOUT`, which is the standard output, and `STDERR` which is the stream for errors.

- `STDIN`: this can be text or binary data streaming into the program, or keyboard input.
- `STDOUT`: this is the stream where the program writes its data. This is printed to the screen unless otherwise specified.

[1]You can probably get away with running them on your local Mac, but beware: the built-in command line tools on OS X (like `mv`) are very old BSD versions. You want to install the new GNU tools using homebrew Then your local Mac environment will be very similar to your remote EC2 environment.

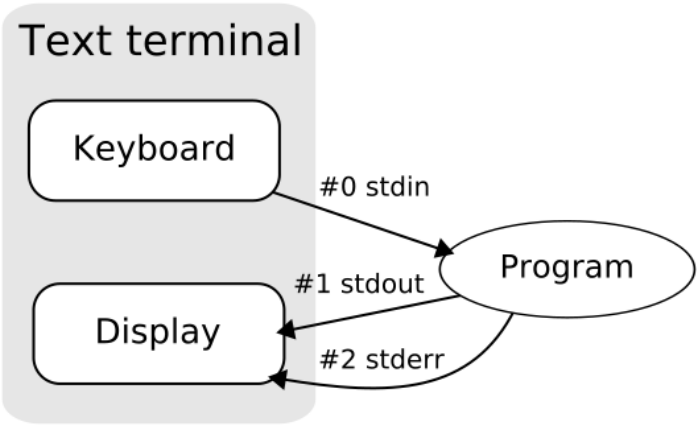

Figure 1: *Visualizing the three standard streams. From Wikipedia.*

- `STDERR`: this is the stream where error messages are display. This is also printed to the screen unless otherwise specified.

Here are a few simple examples; execute these at your EC2 command line.

```
# Redirecting STDOUT with >
echo -e "line1\nline2"
echo -e "line1\nline2" > demo.txt

# Redirecting STDERR with 2>
curl fakeurl # print error message to screen
curl fakeurl 2> errs.txt
cat errs.txt

# Redirecting both STDIN and STDOUT to different files with 1> and 2>
curl google.com fakeurl 1> out1.txt 2> out2.txt

# Redirecting both to the same file with &>
curl google.com fakeurl &> out.txt

# Getting STDIN from a pipe
cat errs.txt | head

# Putting it all together
curl -s http://google.com | head -n 2 &> asdf.txt
```

Note what we did in the last two examples, with the `|` symbol. That's called a pipe, and the technique of connecting one command's STDOUT to another command's STDIN is very powerful. It allows us to build up short-but-powerful programs by composing individual pieces. If you *can* write something this way, especially for text processing or data analysis, you almost always *should*...even if it's a complex ten step pipeline. The reason is that it will be incredibly fast and robust, due to the fact that the underlying GNU tools are written in C

and have received countless hours of optimization and bug fixes. It will also not be that hard to understand: it's still a one-liner, albeit a long one. See this piece on Taco Bell programming and commentary. As a corollary, you should design your own programs to work in command line pipelines (we'll see how to do this as the course progresses). With these principles in mind, let's go through some interactive examples of Unix commands. All of these should work on your default Ubuntu EC2 box.

Navigation and Filesystem

list files: `ls`

The most basic and heavily used command. You will use this constantly to get your bearings and confirm what things do.

```
ls
ls --help
ls --help | less # 'pipe' the output of ls into less
ls -a
ls -alrth
alias ll='ls -alrth' # create an alias in bash
ll
```

modify/create empty files: `touch`

Used to change timestamps and quickly create zero byte files as placeholders. Useful for various kinds of tests.

```
touch file1
touch file2
ll
ll --full-time
touch file1
ll --full-time # note that the file1 modification time is updated
```

display text: `echo`

Useful for debugging and creating small files for tests.

```
echo "foo"
man echo
echo -e "foo\n"
echo -e "line1\nline2" > demo.txt # write directly to tempfile with '>'
```

copy files: `cp`

Powerful command with many options, works on everything from single files to entire archival backups of huge directories.

```
cp --help
cp demo.txt demo2.txt
cp -v demo.txt demo3.txt # verbose copy
cp -a demo.txt demo-archive.txt # archival exact copy
ll --full-time demo* # note timestamps of demo/demo-archive
echo "a new file" > asdf.txt
cat demo.txt
cp asdf.txt demo.txt
cat demo.txt # cp just clobbered the file
alias cp='cp -i' # set cp to be interactive by default
```

move/rename files: `mv`

Move or rename files. Also extremely powerful.

```
mv asdf.txt asdf-new.txt
mv -i asdf-new.txt demo.txt # prompt before clobbering
alias mv='mv -i'
```

remove files: `rm`

Powerful command that can be used to delete individual files or recursively delete entire trees. Use with caution.

```
rm demo.txt
rm -i demo2.txt
rm -i demo*txt
alias rm='rm -i'
alias # print all aliases
```

symbolic link: `ln`

Very useful command that allows a file to be symbolically linked, and therefore in two places at once. Extremely useful for large files, or for putting in one level of indirection.

```
yes | nl | head -1000 > data1.txt
head data1.txt
ll data1.txt
ln -s data1.txt latest.txt
head latest.txt
ll latest.txt # note the arrow. A symbolic link is a pointer.
yes | nl | head -2000 | tail -50 > data2.txt
head latest.txt
ln -s data2.txt latest.txt # update the pointer w/o changing the underlying file
head latest.txt
```

```
head data1.txt
head data2.txt
```

print working directory: `pwd`

Simple command to print current working directory. Used for orientation when you get lost.

```
pwd
```

create directories: `mkdir`

Create directories. Has a few useful options.

```
mkdir dir1
mkdir dir2/subdir    # error
mkdir -p dir2/subdir # ok
```

change current directory: `cd`

Change directories. Along with `ls`, one of the most frequent commands.

```
cd ~
cd dir1
pwd
cd ..
cd dir2/subdir
pwd
cd - # jump back
cd - # and forth
cd   # home
alias ..='cd ..'
alias ...='cd ..; cd ..'
cd dir2/subdir
...
pwd
```

remove directories: `rmdir`

Not used that frequently; usually recursive `rm` is used instead. Note that `rm -rf` is the most dangerous command in Unix and must be used with extreme caution, as it means "remove recursively without prompting" and can easily nuke huge amounts of data.

```
rmdir dir1
rmdir dir2 # error
rm -rf dir2 # works
```

Downloading and Syncing

synchronize local and remote files: `rsync`

Transfer files between two machines. Assuming you have set up your `~/.ssh/config` as in previous lectures, run this command on your *local* machine, making the appropriate substitutions:

```
# Run on local machine
$ yes | nl | head -500 > something.txt
$ rsync -avp something.txt awshost4:~/
building file list ... done
something.txt

sent 4632 bytes  received 42 bytes  1869.60 bytes/sec
total size is 4500  speedup is 0.96
$ rsync -avp something.txt awshost4:~/
building file list ... done

sent 86 bytes  received 20 bytes  42.40 bytes/sec
total size is 4500  speedup is 42.45
```

Notice that the first `rsync` invocation sent 4632 bytes but the second time sent much less data 86 bytes. That is because `rsync` can resume transfers, which is essential for working with large files and makes it generally preferable to `scp`. As an exercise, `ssh` in and run `cat something.txt` on the EC2 machine to see that it was uploaded.

retrieve files via http/https/ftp: `wget`

Very useful command to rapidly pull down files or webpages. Note: the distinction between `wget` and `rsync` is that the `wget` is generally used for publicly accessible files (accessible via a web browser, perhaps behind a login) while `rsync` is used for private files (usually accessible only via `ssh`).

```
wget http://startup-class.s3.amazonaws.com/simple.sh
less simple.sh # hit q to quit

# pull a single HTML page
wget https://github.com/joyent/node/wiki/modules
less modules

# recursively download an entire site, waiting 2 seconds between hits.
wget -w 2 -r -np -k -p http://www.stanford.edu/class/cs106b
```

interact with single URLs: `curl`

A bit different from `wget`, but has some overlap in functionality. `curl` is for interacting with single URLs. It doesn't have the spidering/recursive properties that `wget` has, but it supports

a much wider array of protocols. It is very commonly used as the building block for API calls.

```
curl https://install.meteor.com | less # an example install file
curl -i https://api.github.com/users/defunkt/orgs # a simple API call
GHUSER="defunkt"
GHVAR="orgs"
curl -i https://api.github.com/users/$GHUSER/$GHVAR # with variables
GHVAR="repos"
curl -i https://api.github.com/users/$GHUSER/$GHVAR # with diff vars
```

send test packets: `ping`

See if a remote host is up. Very useful for basic health checks or to see if your computer is online. Has a surprisingly large number of options.

```
ping google.com # see if you have internet connectivity
ping stanford.edu # see if stanford is up
```

Basic Text Processing

view files: `less`

Paging utility used to view large files. Can scroll both up and down. Use `q` to quit.

```
rsync --help | less
```

print/concatenate files: `cat`

Industrial strength file viewer. Use this rather than MS Word for looking at large files, or files with weird bytes.

```
# Basics
echo -e "line1\nline2" > demo.txt
cat demo.txt
cat demo.txt demo.txt demo.txt > demo2.txt
cat demo2.txt

# Piping
yes | head | cat - demo2.txt
yes | head | cat demo2.txt -

# Download chromosome 22 of the human genome
wget ftp://ftp.ncbi.nih.gov/genomes/Homo_sapiens/CHR_22/hs_ref_GRCh37.p10_chr22.gbk.gz
gunzip hs_ref_GRCh37.p10_chr22.gbk.gz
less hs_ref_GRCh37.p10_chr22.gbk
cat hs_ref_GRCh37.p10_chr22.gbk # hit control-c to interrupt
```

first part of files: `head`

Look at the first few lines of a file (10 by default). Surprisingly useful for debugging and inspecting files.

```
head --help
head *gbk      # first 10 lines
head -50 *gbk  # first 50 lines
head -n50 *gbk # equivalent
head *txt *gbk # heads of multiple files
head -q *txt *gbk # heads of multiple files w/o delimiters
head -c50 *gbk # first 50 characters
```

last part of files: `tail`

Look at the bottom of files; default is last 10 lines. Useful for following logs, debugging, and in combination with head to pull out subsets of files.

```
tail --help
tail *gbk
head *gbk
tail -n+3 *gbk | head  # start at third line
head -n 1000 *gbk | tail -n 20 # pull out intermediate section
# run process in background and follow end of file
yes | nl | head -n 10000000 > foo &
tail -F foo
```

extract columns: `cut`

Pull out columns of a file. In combination with head/tail, can pull out arbitrary rectangular subsets of a file. Extremely useful for working with any kind of tabular data (such as data headed for a database).

```
wget ftp://ftp.ncbi.nih.gov/genomes/Bacteria/\
Escherichia_coli_K_12_substr__W3110_uid161931/NC_007779.ptt
head *ptt
cut -f2 *ptt | head
cut -f2,5 *ptt | head
cut -f2,5 *ptt | head -30
cut -f1 *ptt | cut -f1 -d'.' | head
cut -c1-20 *ptt | head
```

numbering lines: `nl`

Number lines. Useful for debugging and creating quick datasets. Has many options for formatting as well.

```
nl *gbk | tail -1 # determine number of lines in file
nl *ptt | tail -2
```

concatenate columns: `paste`

Paste together data by columns.

```
tail -n+3 *ptt | cut -f1 > locs
tail -n+3 *ptt | cut -f5 > genes
paste genes locs genes | head
```

sort by lines: `sort`

Industrial strength sorting command. Very powerful standalone and in combination with others.

```
sort genes | less    # default sort
sort -r genes | less # reverse
sort -R genes | less # randomize
cut -f2 *ptt | tail -n+4 | head
```

uniquify lines: `uniq`

Useful command for analyzing any data with repeated elements. Best used in pipelines with sort beforehand.

```
cut -f2 *ptt | tail -n+4 | sort | uniq -c | sort -k1 -rn #
cut -f3 *ptt | tail -n+4 | uniq -c | sort -k2 -rn # output number
cut -f9 *ptt > products
sort products | uniq -d
```

line, word, character count: `wc`

Determine file sizes. Useful for debugging and confirmation, faster than `nl` if no intermediate information is needed.

```
wc *ptt    # lines, words, bytes
wc -l *ptt # only number of lines
wc -L *ptt # longest line length, useful for apps like style checking
```

split large files: `split`

Split large file into pieces. Useful as initial step before many parallel computing jobs.

```
split -d -l 1000 *ptt subset.ptt.
ll subset.ptt*
```

Help

manuals: `man`

Single page manual files. Fairly useful once you know the basics. But Google or StackOverflow are often more helpful these days if you are really stuck.

```
man bash
man ls
man man
```

info: `info`

A bit more detail than `man`, for some applications.

```
info cut
info info
```

System and Program information

computer information: `uname`

Quick diagnostics. Useful for install scripts.

```
uname -a
```

current computer name: `hostname`

Determine name of current machine. Useful for parallel computing, install scripts, config files.

```
hostname
```

current user: `whoami`

Show current user. Useful when using many different machines and for install scripts.

```
whoami
```

current processes: `ps`

List current processes. Usually a prelude to killing a process.

```
sleep 60 &
ps xw | grep sleep
# kill the process number
```

Here is how that looks in a session:

```
ubuntu@domU-12-31-39-16-1C-96:~$ sleep 20 &
[1] 5483
ubuntu@domU-12-31-39-16-1C-96:~$ ps xw | grep sleep
 5483 pts/0    S      0:00 sleep 20
 5485 pts/0    S+     0:00 grep --color=auto sleep
ubuntu@domU-12-31-39-16-1C-96:~$ kill 5483
[1]+  Terminated              sleep 20
```

So you see that the process number `5483` was identified by printing the list of running processes with `ps xw`, finding the process ID for `sleep 20` via `grep`, and then running `kill 5483`. Note that the `grep` command itself showed up; that is a common occurrence and can be dealt with as follows:

```
ubuntu@domU-12-31-39-16-1C-96:~$ sleep 20 &
[1] 5486
ubuntu@domU-12-31-39-16-1C-96:~$ ps xw | grep sleep | grep -v "grep"
 5486 pts/0    S      0:00 sleep 20
ubuntu@domU-12-31-39-16-1C-96:~$ kill 5486
```

Here, we used the `grep -v` flag to exclude any lines that contained the string `grep`. That will have some false positives (e.g. it will exclude a process named `foogrep`, so use this with some caution.

most important processes: `top`

Dashboard that shows which processes are doing what. Use `q` to quit.

```
top
```

stop a process: `kill`

Send a signal to a process. Usually this is used to terminate misbehaving processes that won't stop of their own accord.

```
sleep 60 &
# The following invocation uses backticks to kill the process we just
# started. Copy/paste it into a separate text file if you can't distinguish the
# backticks from the standard quote.
kill `ps xw | grep sleep | cut -f1 -d ' ' | head -1`
```

Superuser

become root temporarily: `sudo`

Act as the root user for just one or a few commands.

```
sudo apt-get install -y git-core
```

become root: `su`

Actually become the root user. Sometimes you do this when it's too much of a pain to type `sudo` a lot, but you usually don't want to do this.

```
touch normal
sudo su
touch rootfile
ls -alrth normal rootfile # notice owner of rootfile
```

Storage and Finding

create an archive: `tar`

Make an archive of files.

```
mkdir genome
mv *ptt* genome/
tar -cvf genome.tar genome
```

compress files: `gzip`

Compress files. Can also view compressed `gzip` files without fully uncompressing them with `zcat`.

```
gzip genome.tar
md temp
cp genome.tar.gz temp
cd temp
tar -xzvf genome.tar.gz
cd ..
```

```
gunzip genome.tar.gz
rm -rf genome.tar genome temp
wget ftp://ftp.ncbi.nih.gov/genomes/Homo_sapiens/CHR_22/hs_ref_GRCh37.p10_chr22.fa.gz
zcat hs_ref_GRCh37.p10_chr22.fa.gz | nl | head -1000 | tail -20
```

find a file (non-indexed): `find`

Non-indexed search. Can be useful for iterating over all files in a subdirectory.

```
find /etc | nl
```

find a file (indexed): `locate`

For locate to work, `updatedb` must be operational. This runs by default on Ubuntu but you need to manually configure it for a mac.

```
sudo apt-get install -y locate
sudo updatedb # takes a little bit of time
locate fstab
```

system disk space: `df`

Very useful when working with large data sets.

```
df -Th
```

directory utilization: `du`

Use to determine which subdirectories are taking up a lot of disk space.

```
cd ~ubuntu
du --max-depth=1 -b | sort -k1 -rn
```

Intermediate Text Processing

searching within files: `grep`

Powerful command which is worth learning in great detail. Go through `grep --help` and try out many more of the options.

```
wget ftp://ftp.ncbi.nih.gov/genomes/Bacteria/\
Escherichia_coli_K_12_substr__W3110_uid161931/NC_007779.ptt
grep protein *ptt | wc -l # lines containing protein
grep -l Metazoa *ptt *gbk # print filenames which contain 'Metazoa'
grep -B 5 -A 5 Metazoa *gbk
grep 'JOURNAL.*' *gbk | sort | uniq
```

simple substitution: `sed`

Quick find/replace within a file. You should review this outstanding set of useful sed one-liners. Here is a simple example of using `sed` to replace all instances of `kinase` with `STANFORD` in the first 10 lines of a `ptt` file, printing the results to STDOUT:

```
head *ptt | sed 's/kinase/STANFORD/g'
```

`sed` is also useful for quick cleanups of files. Suppose you have a file which was saved on Windows:

```
# Convert windows newlines into unix; use if you have such a file
wget http://startup-class.s3.amazonaws.com/windows-newline-file.csv
```

Viewing this file in `less` shows us some `^M` characters, which are Windows-format newlines (`\r`), different from Unix newlines (`\n`).

```
$ less windows-newline-file.csv
30,johnny@gmail.com,Johnny Walker,,^M1456,jim@gmail.com,Jim Beam,,^M2076,...
windows-newline-file.csv (END)
```

The `\r` is interpreted in Unix as a "carriage return", so you can't just `cat` these files. If you do, then the cursor moves back to the beginning of the line everytime it hits a `\r`, which means you don't see much. Try it out:

```
cat windows-newline-file.csv
```

You won't see anything except perhaps a blur. To understand what is going on in more detail, read this post and this one. We can fix this issue with `sed` as follows:

```
ubuntu@domU-12-31-39-16-1C-96:~$ sed 's/\r/\n/g' windows-newline-file.csv
30,johnny@gmail.com,Johnny Walker,,
1456,jim@gmail.com,Jim Beam,,
2076,jack@stanford.edu,Jack Daniels,,
```

Here `s/\r/\n/g` means "replace the `\r` with `\n` globally" in the file. Without the trailing `g` the `sed` command would just replace the first match. Note that by default `sed` just writes its output to STDOUT and does not modify the underlying file; if we actually want to modify the file in place, we would do this instead:

```
sed -i 's/\r/\n/g' windows-newline-file.csv
```

Note the `-i` flag for in-place replacement. Again, it is worth reviewing this list of useful sed one-liners.

advanced substitution/short scripts: `awk`

Useful scripting language for working with tab-delimited text. Very fast for such purposes, intermediate size tool.

```
tail -n+4 *ptt | awk -F"\t" '{print $2, $3, $3 + 5}' | head
```

To get a feel for `awk`, review this list of useful awk one liners. It is often the fastest way to operate on tab-delimted data before importation into a database.

Intermediate `bash`

Keyboard shortcuts

Given how much time you will spend using `bash`, it's important to learn all the keyboard shortcuts.

- *Ctrl* + *A*: Go to the beginning of the line you are currently typing on
- *Ctrl* + *E*: Go to the end of the line you are currently typing on
- *Ctrl* + *F*: Forward one character.
- *Ctrl* + *B*: Backward one character.
- *Meta* + *F*: Move cursor forward one word on the current line
- *Meta* + *B*: Move cursor backward one word on the current line
- *Ctrl* + *P*: Previous command entered in history
- *Ctrl* + *N*: Next command entered in history
- *Ctrl* + *L*: Clears the screen, similar to the `clear` command
- *Ctrl* + *U*: Clears the line before the cursor position. If you are at the end of the line, clears the entire line.
- *Ctrl* + *H*: Same as backspace
- *Ctrl* + *R*: Lets you search through previously used commands
- *Ctrl* + *C*: Kill whatever you are running
- *Ctrl* + *D*: Exit the current shell
- *Ctrl* + *Z*: Puts whatever you are running into a suspended background process. `fg` restores it.
- *Ctrl* + *W*: Delete the word before the cursor
- *Ctrl* + *K*: Kill the line after the cursor
- *Ctrl* + *Y*: Yank from the kill ring

- *Ctrl* + _: Undo the last bash action (e.g. a yank or kill)
- *Ctrl* + *T*: Swap the last two characters before the cursor
- *Meta* + *T*: Swap the last two words before the cursor
- *Tab*: Auto-complete files and folder names

Let's go through a brief screencast of the most important shortcuts.

Backticks

Sometimes you want to use the results of a bash command as input to another command, but not via STDIN. In this case you can use backticks:

```
echo `hostname`
echo "The date is "`date`
```

This is a useful technique to compose command line invocations when the results of one is the argument for another. For example, one command might return a hostname like `ec2-50-17-88-215.compute-1.amazonaws.com` which can then be passed in as the `-h` argument for another command via backticks.

Running processes in the background: `foo &`

Sometimes we have a long running process, and we want to execute other commands while it completes. We can put processes into the background with the ampersand symbol as follows.

```
sleep 50 &
# do other things
```

Here is how that actually looks at the command line:

```
ubuntu@domU-12-31-39-16-1C-96:~$ sleep 60 &
[1] 5472
ubuntu@domU-12-31-39-16-1C-96:~$ head -2 *ptt
Escherichia coli str. K-12 substr. W3110, complete genome - 1..4646332
4217 proteins
ubuntu@domU-12-31-39-16-1C-96:~$ ps xw | grep 5472
 5472 pts/0    S      0:00 sleep 60
 5475 pts/0    S+     0:00 grep --color=auto 5472
ubuntu@domU-12-31-39-16-1C-96:~$
ubuntu@domU-12-31-39-16-1C-96:~$
[1]+  Done                    sleep 60
```

Note that we see `[1] 5472` appear. That means there is one background process currently running (`[1]`) and that the ID of the background process we just spawned is `5472`. Note also at the end that when the process is done, this line appears:

```
[1]+  Done                    sleep 60
```

That indicates which process is done. For completeness, here is what it would look like if you had multiple background processes running simultaneously, and then waited for them all to end.

```
ubuntu@domU-12-31-39-16-1C-96:~$ sleep 10 &
[1] 5479
ubuntu@domU-12-31-39-16-1C-96:~$ sleep 20 &
[2] 5480
ubuntu@domU-12-31-39-16-1C-96:~$ sleep 30 &
[3] 5481
ubuntu@domU-12-31-39-16-1C-96:~$
[1]   Done                    sleep 10
[2]-  Done                    sleep 20
[3]+  Done                    sleep 30
```

Execute commands from STDIN: `xargs`

This is a very powerful command but a bit confusing. It allows you to programmatically build up command lines, and can be useful for spawning parallel processes. It is commonly used in combination with the `find` or `ls` commands. Here is an example which lists all files under `/etc` ending in `.sh`, and then invokes `head -2` on each of them.

```
find /etc -name '*\.sh'   | xargs head -2
```

Here is another example which invokes the `file` command on everything in `/etc/profile.d`, to print the file type:

```
ls /etc/profile.d | xargs file
```

A savvy reader will ask why one couldn't just do this:

```
file /etc/profile.d/*
```

Indeed, that will produce the same results in this case, and is feasible because only one directory is being listed. However, if there are a huge number of files in the directory, then `bash` will be unable to expand the `*`. This is common in many large-scale applications in search, social, genomics, etc and will give the dreaded `Argument list too long` error. In this case `xargs` is the perfect tool; see here for more.

Pipe and redirect: `tee`

Enables you to save intermediate stages in a pipeline. Here's an example:

```
ls | tee list.txt
```

The reason it is called `tee` is that it is like a "T-junction", where it passes the data through while also serializing it to a file. It can be useful for backing up a file while simultaneously modifying it; see some examples.

time any command: `time`

This is a `bash` built in useful for benchmarking commands.

```
time sleep 5
```

That should echo the fact that the command took (almost) exactly 5 seconds to complete. It is also useful to know about a more sophisticated `GNU time` command, which can output a ton of useful information, including maximum memory consumption; extremely useful to know before putting a job on a large compute cluster. You can invoke it with `/usr/bin/time`; see documentation for Ubuntu 12 LTS here.

Summary

We covered quite a few command line programs here, but nothing substitutes for running them on your own and looking at the online help and flags. For example, with `head` you can do `head --help`, `man head`, and `info head` to get more information.