100 Useful Command-Line Utilities

A Guide to 100 (ish) Useful Commands on Unix-like systems

by Oliver; 2014

command_line bash unix linux learn_command_line command_line_utilities

Table of Contents

44. gzip, gunzip, bzip2, bunzip2 VIEW_AS_PAGE
45. tar VIEW_AS_PAGE
46. uniq VIEW_AS_PAGE
47. dirname, basename VIEW_AS_PAGE
48. set, unset VIEW_AS_PAGE
49. env VIEW_AS_PAGE
50. uname VIEW_AS_PAGE
51. df, du VIEW_AS_PAGE
52. bind VIEW_AS_PAGE
53. alias, unalias VIEW_AS_PAGE
54. column VIEW_AS_PAGE
55. find VIEW_AS_PAGE
56. touch VIEW_AS_PAGE
57. diff, comm VIEW_AS_PAGE
58. join VIEW_AS_PAGE
59. md5, md5sum VIEW_AS_PAGE
60. tr VIEW_AS_PAGE
61. od VIEW_AS_PAGE
62. split VIEW_AS_PAGE
63. nano, emacs, vim VIEW_AS_PAGE
64. tree VIEW_AS_PAGE
65. screen VIEW_AS_PAGE
66. tmux VIEW_AS_PAGE
67. make VIEW_AS_PAGE
68. yes VIEW_AS_PAGE
69. nl VIEW_AS_PAGE
70. whoami VIEW_AS_PAGE
71. groups VIEW_AS_PAGE
72. who, w VIEW_AS_PAGE
73. hostname VIEW_AS_PAGE
74. finger VIEW_AS_PAGE
75. read VIEW_AS_PAGE
76. tee VIEW_AS_PAGE
77. shopt VIEW_AS_PAGE
78. true, false VIEW_AS_PAGE
79. shift VIEW_AS_PAGE
80. g++ VIEW_AS_PAGE
81. xargs VIEW_AS_PAGE
82. crontab VIEW_AS_PAGE
83. type VIEW_AS_PAGE
84. info VIEW_AS_PAGE
85. apropos VIEW_AS_PAGE
86. fold VIEW_AS_PAGE
87. rev VIEW_AS_PAGE
88. mount VIEW_AS_PAGE
89. mktemp VIEW_AS_PAGE
90. watch VIEW_AS_PAGE
91. perl, python VIEW_AS_PAGE
92. ping VIEW_AS_PAGE
93. dig VIEW_AS_PAGE

Introduction TOP VIEW_AS_PAGE

In An Introduction to the Command-Line (on Unix-like systems) [1], which I'll borrow heavily from in this article, we covered my subjective list of the 20 most important command line utilities. In case you forgot, it was:

```
pwd
ls
cd
mkdir
echo
cat
cp
mv
rm
man
head
tail
less
more
sort
grep
which
chmod
history
clear
```

These are indispensable but, for anything more than the basics, a vocabulary of about 100 commands is where you want to be. Here are my picks for the top 100(ish) commands. I'll briefly introduce each one below along with a quick use case or hint.

A few of the commands, mostly clustered at the end of the article, are not standard shell commands and you have to install them yourself (see `apt-get`, `brew`, `yum`). When this is the case, I make a note of it.

---

[1] Note: Unix-like systems include Linux and Macintosh ↑

pwd TOP VIEW_AS_PAGE

To see where we are, we can *print working directory*, which returns the path of the directory in which we currently reside:

```
$ pwd
```

Sometimes in a script we change directories, but we want to save the directory in which we started. We can save the current working directory in a variable with command substitution:

```
d=$( pwd )              # save cwd
cd /somewhere/else      # go somewhere else
                        # do something else
cd $d                   # go back to where the script originally started
```

If you want to print your working directory resolving symbolic links:

```
$ readlink -m .
```

cd TOP VIEW_AS_PAGE

To move around, we can *change directory*:

```
$ cd /some/path
```

By convention, if we leave off the argument and just type `cd` we go `HOME`:

```
$ cd     # go $HOME
```

Go to the root directory:

```
$ cd /   # go to root
```

Move one directory back:

```
$ cd ..  # move one back toward root
```

To return to the directory we were last in:

```
$ cd -  # cd into the previous directory
```

What if you want to visit the directory you were in two or three directories ago? The utilities `pushd` and `popd` keep track of the directories you move through in a stack. When you use:

```
$ pushd some_directory
```

It acts as a:

```
$ cd some_directory
```

except that *some_directory* is also added to the stack. Let's see an example of how to use this:

```
$ pushd ~/TMP           # we're in ~/
~/TMP ~
$ pushd ~/DATA          # now we're in ~/TMP
~/DATA ~/TMP ~
$ pushd ~               # now we're in ~/DATA
~ ~/DATA ~/TMP ~
$ popd                  # now we're in ~/
~/DATA ~/TMP ~
$ popd                  # now we're in ~/DATA
~/TMP ~
$ popd                  # now we're in ~/TMP
~
$                       # now we're in ~/
```

This is interesting to see once, but I never use `pushd` and `popd`. Instead, I like to use a function, *cd_func*, which I stole off the internet. It allows you to scroll through all of your past directories with the syntax:

```
$ cd --
```

Here's the function, which you should paste into your setup dotfiles (e.g., *.bash_profile*):

```
cd_func ()
{
    local x2 the_new_dir adir index;
    local -i cnt;
```

```
    if [[ $1 == "--" ]]; then
        dirs -v;
        return 0;
    fi;
    the_new_dir=$1;
    [[ -z $1 ]] && the_new_dir=$HOME;
    if [[ ${the_new_dir:0:1} == '-' ]]; then
        index=${the_new_dir:1};
        [[ -z $index ]] && index=1;
        adir=$(dirs +$index);
        [[ -z $adir ]] && return 1;
        the_new_dir=$adir;
    fi;
    [[ ${the_new_dir:0:1} == '~' ]] && the_new_dir="${HOME}${the_new_dir:1}";
    pushd "${the_new_dir}" > /dev/null;
    [[ $? -ne 0 ]] && return 1;
    the_new_dir=$(pwd);
    popd -n +11 2> /dev/null > /dev/null;
    for ((cnt=1; cnt <= 10; cnt++))
    do
        x2=$(dirs +${cnt} 2>/dev/null);
        [[ $? -ne 0 ]] && return 0;
        [[ ${x2:0:1} == '~' ]] && x2="${HOME}${x2:1}";
        if [[ "${x2}" == "${the_new_dir}" ]]; then
            popd -n +$cnt 2> /dev/null > /dev/null;
            cnt=cnt-1;
        fi;
    done;
    return 0
}
```

Also in your setup dotfiles, alias `cd` to this function:

```
alias cd=cd_func
```

and then you can type:

```
$ cd --
 0  ~
 1  ~/DATA
 2  ~/TMP
```

to see all the directories you've visited. To go into *~/TMP*, for example, enter:

```
$ cd -2
```

Changing the subject and dipping one toe into the world of shell scripting, you can put `cd` into constructions like this:

```
if ! cd $outputdir; then echo "error. couldn't cd into "${outputdir}; exit; fi
```

or, more succinctly:

```
cd $outputdir || exit
```

This can be a useful line to include in a script. If the user gives an output directory as an argument and the directory doesn't exist, we exit. If it does exist, we `cd` into it and it's business as usual.

ls TOP VIEW_AS_PAGE

`ls` lists the files and directories in the *cwd*, if we leave off arguments. If we pass directories to the command as arguments, it will list their contents. Here are some common flags.

Display our files in a column:

```
$ ls -1  # list vertically with one line per item
```

List in long form—show file permissions, the owner of the file, the group to which he belongs, the date the file was created, and the file size:

```
$ ls -l  # long form
```

List in *human-readable* (bytes will be rounded to kilobytes, gigabytes, etc.) long form:

```
$ ls -hl  # long form, human readable
```

List in human-readable long form sorted by the time the file was last modified:

```
$ ls -hlt # long form, human readable, sorted by time
```

List in long form *all* files in the directory, including *dotfiles*:

```
$ ls -al  # list all, including dot- files and dirs
```

Since you very frequently want to use `ls` with these flags, it makes sense to `alias` them:

```
alias ls="ls --color=auto"
alias l="ls -hl --color=auto"
alias lt="ls -hltr --color=auto"
alias ll="ls -al --color=auto"
```

The coloring is key. It will color directories and files and executables differently, allowing you to quickly scan the contents of your folder.

Note that you can use an arbitrary number of arguments and that bash uses the convention that an asterik matches anything. For example, to list only files with the *.txt* extension:

```
$ ls *.txt
```

This is known as globbing. What would the following do?

```
$ ls . dir1 .. dir2/*.txt dir3/A*.html
```

This monstrosity would list anything in the *cwd*; anything in directory *dir1*; anything in the directory one above us; anything in directory *dir2* that ends with *.txt*; and anything in directory *dir3* that starts with *A* and ends with *.html*. You get the point!

`ls` can be useful in for-loops:

```
$ for i in $( ls /some/path/*.txt ); do echo $i; done
```

With this construct, we could do some series of commands on all *.txt* files in */some/path*. You can also do a very similar thing without `ls`:

```
$ for i in /some/path/*.txt; do echo $i; done
```

mkdir, rmdir TOP VIEW_AS_PAGE

To *make directory*—i.e., create a new folder—we use:

```
$ mkdir mynewfolder
```

To make nested directories (and don't complain if trying to make a directory that already exists), use the `-p` flag:

```
$ mkdir -p a/b/c  # make nested directories
```

You can *remove directory* using:

```
$ rmdir mynewfolder  # don't use this
```

However, since the directory must be empty to use this command, it's not convenient. Instead, use:

```
$ rm -r mynewfolder  # use this
```

Since `rm` has all the functionality you need, I know few people who actually use `rmdir`. About the only occasion to use it is if you want to be careful you're not deleting a directory with stuff in it. (Perhaps `rmdir` isn't one of the 100 most useful commands, after all.)

echo TOP VIEW_AS_PAGE

`echo` prints the *string* passed to it as an argument. For example:

```
$ echo joe
joe
```

```
$ echo "joe"
joe
```

```
$ echo "joe joe"
joe joe
```

If you leave off an argument, echo will produce a newline:

```
$ echo

```

Supress newline:

```
$ echo -n "joe"            # suppress newline
```

Interpret special characters:

```
$ echo -e "joe\tjoe\njoe"  # interpret special chars ( \t is tab, \n newline )
joe     joe
joe
```

As we've seen above, if you want to print a string with spaces, use quotes. You should also be aware of how bash treats double vs single quotes. If you use double quotes, any variable inside them will be expanded (the same as in Perl). If you use single quotes, everything is taken literally and variables are not expanded. Here's an example:

```
$ var=5
$ joe=hello $var
-bash: 5: command not found
```

That didn't work because we forgot the quotes. Let's fix it:

```
$ joe="hello $var"
$ echo $joe
hello 5
```

```
$ joe='hello $var'
$ echo $joe
hello $var
```

Sometimes, when you run a script verbosely, you want to echo commands before you execute them. A std:out log file of this type is invaluable if you want to retrace your steps later. One way of doing this is to save a command in a variable, *cmd*, echo it, and then pipe it into `bash` or `sh`. Your script might look like this:

```
cmd="ls -hl";
echo $cmd;
echo $cmd | bash
```

cat, zcat, tac TOP VIEW_AS_PAGE

`cat` prints the *contents* of files passed to it as arguments. For example:

```
$ cat file.txt
```

prints the contents of *file.txt*. Entering:

```
$ cat file.txt file2.txt
```

would print out the contents of both *file.txt* and *file2.txt* con**cat**enated together, which is where this command gets its slightly confusing name. Print file with line numbers:

```
$ cat -n file.txt       # print file with line numbers
```

`cat` is frequently seen in unix pipelines. For instance:

```
$ cat file.txt | wc -l          # count the number of lines in a file
$ cat file.txt | cut -f1        # cut the first column
```

Some people deride this as unnecessarily verbose, but I'm so used to piping anything and everything that I embrace it. Another common construction is:

```
cat file.txt | awk ...
```

We'll discuss `awk` below, but the key point about it is that it works line by line. So `awk` will process what `cat` pipes out in a linewise fashion.

If you route something to `cat` via a pipe, it just passes through:

```
$ echo "hello kitty" | cat
hello kitty
```

The `-vet` flag allows us to "see" special characters, like tab, newline, and carriage return:

```
$  echo -e "\t" | cat -vet
^I$
$ echo -e "\n" | cat -vet
$
$
$ echo -e "\r" | cat -vet
^M$
```

This can come into play if you're looking at a file produced on a PC, which uses the horrid *\r* at the end of a line as opposed to the nice unix newline, *\n*. You can do a similar thing with the command `od`, as we'll see below .

There are two variations on `cat`, which occasionally come in handy. `zcat` allows you to `cat` zipped files:

```
$ zcat file.txt.gz
```

You can also see a file in reverse order, bottom first, with `tac` (*tac* is *cat* spelled backwards).

cp TOP VIEW_AS_PAGE

The command to make a copy of a file is `cp`:

```
$ cp file1 file2
```

Use the recursive flag, *-R*, to copy directories plus files:

```
$ cp -R dir1 dir2  # copy directories
```

The directory and everything inside it are copied.

Question: what would the following do?

```
$ cp -R dir1 ../../
```

Answer: it would make a copy of *dir1* up two levels from our current working directory.

Tip: If you're moving a large directory structure with lots of files in them, use `rsync` instead of `cp`. If the command fails midway through, `rsync` can start from where it left off but `cp` can't.

mv TOP VIEW_AS_PAGE

To rename a file or directory we use `mv`:

```
$ mv file1 file2
```

In a sense, this command also moves files, because we can rename a file into a different path. For example:

```
$ mv file1 dir1/dir2/file2
```

would move *file1* into *dir1/dir2/* and change its name to *file2*, while:

```
$ mv file1 dir1/dir2/
```

would simply move *file1* into *dir1/dir2/* (or, if you like, rename *./file1* as *./dir1/dir2/file1*).

Swap the names of two files, *a* and *b*:

```
$ mv a a.1
$ mv b a
$ mv a.1 b
```

Change the extension of a file, *test.txt*, from *.txt* to *.html*:

```
$ mv test.txt test.html
```

Shortcut:

```
$ mv test.{txt,html}
```

`mv` can be dangerous because, if you move a file into a directory where a file of the same name exists, the latter will be overwritten. To prevent this, use the *-n* flag:

```
$ mv -n myfile mydir/ # move, unless "myfile" exists in "mydir"
```

rm TOP VIEW_AS_PAGE

The command `rm` removes the files you pass to it as arguments:

```
$ rm file      # removes a file
```

Use the recursive flag, *-r*, to remove a file or a directory:

```
$ rm -r dir    # removes a file or directory
```

If there's a permission issue or the file doesn't exist, `rm` will throw an error. You can override this with the force flag, *-f*:

```
$ rm -rf dir   # force removal of a file or directory
               # (i.e., ignore warnings)
```

You may be aware that when you delete files, they can still be recovered with effort if your computer hasn't overwritten their contents. To securely delete your files—meaning overwrite them before deleting—use:

```
$ rm -P file   # overwrite your file then delete
```

or use `shred`.

shred TOP VIEW_AS_PAGE

Securely remove your file by overwritting then removing:

```
$ shred -zuv file        # removes a file securely
                         # (flags are: zero, remove, verbose)
```

For example:

```
$ touch crapfile  # create a file
$ shred -zuv crapfile
shred: crapfile: pass 1/4 (random)...
shred: crapfile: pass 2/4 (random)...
shred: crapfile: pass 3/4 (random)...
shred: crapfile: pass 4/4 (000000)...
shred: crapfile: removing
shred: crapfile: renamed to 00000000
shred: 00000000: renamed to 0000000
shred: 0000000: renamed to 000000
shred: 000000: renamed to 00000
shred: 00000: renamed to 0000
shred: 0000: renamed to 000
shred: 000: renamed to 00
shred: 00: renamed to 0
shred: crapfile: removed
```

As the man pages note, this isn't perfectly secure if your file system has made a copy of your file and stored it in some other location. Read the man page here.

man TOP VIEW_AS_PAGE

`man` shows the usage manual, or help page, for a command. For example, to see the manual for `ls`:

```
$ man ls
```

To see `man`'s own manual page:

```
$ man man
```

The manual pages will list all of the command's flags which usually come in a *one-dash-one-letter* or *two-dashes-one-word* flavor:

```
command -f
command --flag
```

A related command discussed below is `info`.

head, tail TOP VIEW_AS_PAGE

Based off of An Introduction to the Command-Line (on Unix-like systems) - head and tail: `head` and `tail` print the first or last *n* lines of a file, where *n* is 10 by default. For example:

```
$ head myfile.txt      # print the first 10 lines of the file
$ head -1 myfile.txt   # print the first line of the file
$ head -50 myfile.txt  # print the first 50 lines of the file
```

```
$ tail myfile.txt      # print the last 10 lines of the file
$ tail -1 myfile.txt   # print the last line of the file
$ tail -50 myfile.txt  # print the last 50 lines of the file
```

These are great alternatives to `cat`, because often you don't want to spew out a giant file. You only want to peek at it to see the formatting, get a sense of how it looks, or hone in on some specific portion.

If you combine `head` and `tail` together in the same command chain, you can get a specific row of your file by row number. For example, print row 37:

```
$ cat -n file.txt | head -37 | tail -1  # print row 37
```

Another thing I like to do with `head` is look at multiple files simultaneously. Whereas `cat` concatencates files together, as in:

```
$ cat hello.txt
hello
hello
hello
```

```
$ cat kitty.txt
kitty
kitty
kitty
```

```
$ cat hello.txt kitty.txt
hello
hello
hello
kitty
kitty
kitty
```

`head` will print out the file names when it takes multiple arguments, as in:

```
$ head -2 hello.txt kitty.txt
==> hello.txt <==
hello
hello

==> kitty.txt <==
kitty
kitty
```

which is useful if you're previewing many files. To preview all files in the current directory:

```
$ head *
```

See the last 10 lines of your bash history:

```
$ history | tail        # show the last 10 lines of history
```

See the first 10 elements in the cwd:

```
$ ls | head
```

less, zless, more TOP VIEW_AS_PAGE

Based off of An Introduction to the Command-Line (on Unix-like systems) - less and more: `less` is, as the *man* pages say, *"a filter for paging through text one screenful at a time...which allows backward movement in the file as well as forward movement."* This makes it one of the odd unix commands whose name seems to bear no relation to its function. If you have a big file, vanilla `cat` is not suitable because printing thousands of lines of text to stdout will flood your screen. Instead, use `less`, the go-to command for viewing files in the terminal:

```
$ less myfile.txt       # view the file page by page
```

Another nice thing about `less` is that it has many Vim-like features, which you can read about on its *man* page (and this is not a coincidence). For example, if you want to search for the word *apple* in your file, you just type slash (`/`) followed by *apple*.

If you have a file with many columns, it's hard to view in the terminal. A neat `less` flag to solve this problem is *-S*:

```
$ less -S myfile.txt     # allow horizontal scrolling
```

This enables horizontal scrolling instead of having rows messily wrap onto the next line. As we'll discuss below, this flag works particularly well in combination with the `column` command, which forces the columns of your file to line up nicely:

```
cat myfile.txt | column -t | less -S
```

Use `zless` to less a zipped file:

```
$ zless myfile.txt.gz     # view a zipped file
```

I included `less` and `more` together because I think about them as a pair, but I told a little white lie in calling `more` indispensible: you really only need `less`, which is an improved version of `more`. Less is more :-)

grep, egrep TOP VIEW_AS_PAGE

Based off of An Introduction to the Command-Line (on Unix-like systems) - grep: `grep` is the terminal's analog of *find* from ordinary computing (not to be confused with unix `find`). If you've ever used Safari or TextEdit or Microsoft Word, you can find a word with ⌘F (*Command-f*) on Macintosh. Similarly, `grep` searches for text in a file and returns the line(s) where it finds a match. For example, if you were searching for the word *apple* in your file, you'd do:

```
$ grep apple myfile.txt          # return lines of file with the text apple
```

`grep` has many nice flags, such as:

```
$ grep -n apple myfile.txt       # include the line number
$ grep -i apple myfile.txt       # case insensitive matching
$ grep --color apple myfile.txt  # color the matching text
```

Also useful are what I call the *ABCs of Grep*—that's *A*fter, *B*efore, *C*ontext. Here's what they do:

```
$ grep -A1 apple myfile.txt      # return lines with the match,
                                 # as well as 1 after
```

```
$ grep -B2 apple myfile.txt      # return lines with the match,
                                 # as well as 2 before
```

```
$ grep -C3 apple myfile.txt      # return lines with the match,
                                 # as well as 3 before and after.
```

You can do an inverse grep with the *-v* flag. Find lines that *don't contain* apple:

```
$ grep -v apple myfile.txt       # return lines that don't contain apple
```

Find any occurrence of apple in any file in a directory with the recursive flag:

```
$ grep -R apple mydirectory/     # search for apple in any file in mydirectory
```

`grep` works nicely in combination with `history`:

```
$ history | grep apple  # find commands in history containing "apple"
```

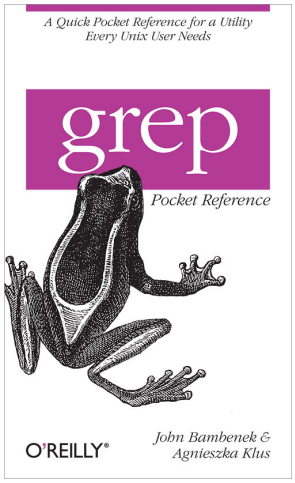
Exit after, say, finding the first two instances of *apple* so you don't waste more time searching:

```
$ cat myfile.txt | grep -m 2 apple
```

There are more powerful variants of grep, like `egrep`, which permits the use of regular expressions, as in:

```
$ egrep "apple|orange" myfile.txt  # return lines with apple OR orange
```

as well as other fancier grep-like tools, such as ack, available for download. For some reason—perhaps because it's useful in a quick and dirty sort of a way and doesn't have any other meanings in English—grep has inspired a peculiar cult following. There are grep t-shirts, grep memes, a grep function in Perl, and—unbelievably—even a whole O'Reilly book devoted to the command:



(Image credit: O'Reilly Media)

Tip: If you're a Vim user, running `grep` as a system command within Vim is a neat way to filter text. Read more about it in Wiki Vim - System Commands in Vim.

Tip: Recently a grep-like tool called fzf has become popular. The authors call it a "*general-purpose command-line fuzzy finder*".

which TOP VIEW_AS_PAGE

`which` shows you the path of a command in your `PATH`. For example, on my computer:

```
$ which less
/usr/bin/less
```

```
$ which cat
/bin/cat
```

```
$ which rm
/bin/rm
```

If there is more than one of the same command in your `PATH`, `which` will show you the one *which* you're using (i.e., the first one in your `PATH`). Suppose your `PATH` is the following:

```
$ echo $PATH
/home/username/mydir:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin
```

And suppose *myscript.py* exists in 2 locations in your `PATH`:

```
$ ls /home/username/mydir/myscript.py
/home/username/mydir/myscript.py
```

```
$ ls /usr/local/bin/myscript.py
/usr/local/bin/myscript.py
```

Then:

```
$ which myscript.py
/home/username/mydir/myscript.py  # this copy of the script has precedence
```

You can use `which` in if-statements to test for dependencies:

```
if ! which mycommand > /dev/null; then echo "command not found"; fi
```

My friend, Albert, wrote a nice tool called *catwhich*, which `cat`s the file returned by `which`:

```
#!/bin/bash

# cat a file in your path

file=$(which $1 2>/dev/null)

if [[ -f $file ]]; then
    cat $file
else
    echo "file \"$1\" does not exist!"
fi
```

This is useful for reading files in your `PATH` without having to track down exactly what directory you put them in.

chmod TOP VIEW_AS_PAGE

From An Introduction to the Command-Line (on Unix-like systems) - chmod: `chmod` adds or removes permissions from files or directories. In unix there are three spheres of permissions:

u - user

g - group

o - other/world

Everyone with an account on the computer is a unique user (see `whoami`) and, although you may not realize it, can be part of various groups, such as a particular lab within a university or a team in a company. There are also three types of permission:

r - read

w - write

x - execute

Recall that we can see a file's permissions by listing it:

```
$ ls -hl myfile
```

We can mix and match permission types and entities how we like, using a plus sign to grant permissions according to the syntax:

```
chmod entity+permissiontype
```

or a minus sign to remove permissions:

```
chmod entity-permissiontype
```

E.g.:

```
$ chmod u+x myfile      # make executable for you
$ chmod g+rxw myfile    # add read write execute permissions for the group
$ chmod go-wx myfile    # remove write execute permissions for the group
                        # and for everyone else (excluding you, the user)
```

You can also use *a* for "all of the above", as in:

```
$ chmod a-rwx myfile    # remove all permissions for you, the group,
                        # and the rest of the world
```

If you find the above syntax cumbersome, there's a numerical shorthand you can use with `chmod`. The only three I have memorized are *000*, *777*, and *755*:

```
$ chmod 000 myfile      # revoke all permissions (---------)
$ chmod 777 myfile      # grant all permissions (rwxrwxrwx)
$ chmod 755 myfile      # reserve write access for the user,
                        # but grant all other permissions (rwxr-xr-x)
```

Read more about the numeric code here. In general, it's a good practice to allow your files to be writable by you alone, unless you have a compelling reason to share access to them.

chown TOP VIEW_AS_PAGE

`chown`, less commonly seen than its cousin `chmod`, changes the owner of a file. As an example, suppose *myfile* is owned by *root*, but you want to grant ownership to *ubuntu* (the default user on ubuntu) or *ec2-user* (the default user on Amazon Linux). The syntax is:

```
$ sudo chown ec2-user myfile
```

If you want to change both the user to *someuser* and the group to *somegroup*, use:

```
$ sudo chown someuser:somegroup myfile
```

If you want to change the user and group to your current user, `chown` works well in combination with `whoami`:

```
$ sudo chown $( whoami ):$( whoami ) myfile
```

history TOP VIEW_AS_PAGE

From An Introduction to the Command-Line (on Unix-like systems) - history: Bash is a bit like the NSA in that it's secretly recording everything you do—at least on the command line. But fear not: unless you're doing something super unsavory, you want your history preserved because having access to previous commands you've entered in the shell saves a lot of labor. You can see your history with the command `history`:

```
$ history
```

The easiest way to search the history, as we've seen, is to bind the Readline Function *history-search-backward* to something convenient, like the up arrow. Then you just press up to scroll backwards through your history. So, if you enter a *c*, pressing up will step through all the commands you entered that began with *c*. If you want to find a command that contains the word *apple*, a useful tool is *reverse intelligent search*, which is invoked with *Cntrl-r*:

```
(reverse-i-search)`':
```

Although we won't cover *unix pipelines* until the next section, I can't resist giving you a sneak-preview. An easier way to find any commands you entered containing *apple* is:

```
$ history | grep apple
```

If you wanted to see the last 10 commands you entered, it would be:

```
$ history | tail
```

What's going on under the hood is somewhat complicated. Bash actually records your command history in two ways: *(1)* it stores it in memory—the *history list*—and *(2)* it stores it in a file—the *history file*. There are a slew of global variables that mediate the behavior of history, but some important ones are:
`HISTFILE` - the path to the history file
`HISTSIZE` - how many commands to store in the history list (memory)
`HISTFILESIZE` - how many commands to store in the history file
Typically, `HISTFILE` is a dotfile residing in `HOME`:

```
~/.bash_history
```

It's important to understand the interplay between the history list and the history file. In the default setup, when you type a command at the prompt it goes into your history list—and is thus revealed via `history`—but it doesn't get written into your history file until you log out. The next time you log in, the contents of your history file are automatically read into memory (your history list) and are thus searchable in your session. So, what's the difference between the following two commands?

```
$ history | tail
$ tail ~/.bash_history
```

The former will show you the last 10 commands you entered in your current session while the later will show you last 10 commands from the previous session. This is all well and good, but let's imagine the following scenario: you're like me and you have 5 or 10 different terminal windows open at the same time. You're constantly switching windows and entering commands, but history-search is such a useful function that you want any command you enter in any one of the 5 windows to be immediately accessible on all the others. We can accomplish this by putting the following lines in our setup dotfile (*.bash_profile*):

```
# append history to history file as you type in commands
shopt -s histappend
export PROMPT_COMMAND='history -a'
```

How does this bit of magic work? Here's what the *histappend* option does, to quote from the man page of `shopt`:

> If set, the history list is appended to the history file when the shell exits, rather than overwriting the history file.
> shopt -s histappend
> To append every line to history individually set:
> PROMPT_COMMAND='history -a'
> With these two settings, a new shell will get the history lines from all previous shells instead of the default 'last window closed'>history (the history file is named by the value of the HISTFILE variable)

Let's break this down. `shopt` stands for "shell options" and is for enabling and disabling various miscellaneous options. The

```
history -a
```

part will *"Append the new history lines (history lines entered since the beginning of the current Bash session) to the history file"* (as the man page says). And the global variable `PROMPT_COMMAND` is a rather funny one that executes whatever code is stored in it right before each printing of the prompt. Put these together and you're immediately updating the *~/.bash_history* file with every command instead of waiting for this to happen at logout. Since every shell can contribute in real time, you can run multiple shells in parallel and have access to a common history on all of them—a good situation.

What if you are engaged in unsavory behavior and want to cover your tracks? You can clear your history as follows:

```
$ history -c  # clear the history list in memory
```

However, this only deletes what's in memory. If you really want to be careful, you had better check your history file, *.bash_history*, and delete any offending portions. Or just wipe the whole thing:

```
$ echo > ~/.bash_history
```

You can read more about history on gnu.org.

A wise man once said, *"Those who don't know history are destined to [re-type] it."* Often it's necessary to retrace your footsteps or repeat some commands. In particular, if you're doing research, you'll be juggling lots of scripts and files and how you produced a file can quickly become mysterious. To deal with this, I like to selectively keep shell commands worth remembering in a "notes" file. If you looked on my computer, you'd see *notes* files scattered everywhere. You might protest: but these are in your history, aren't they? Yes, they are, but the history is gradually erased; it's clogged with trivial commands like `ls` and `cd`; and it doesn't have information about where the command was run. A month from now, your *~/.bash_history* is unlikely to have that long command you desperately need to remember, but your curated *notes* file will.

To make this easy I use the following alias:

```
alias n="history | tail -2 | head -1 | tr -s ' ' | cut -d' ' -f3- | awk '{prin
```

or equivalently:

```
alias n="echo -n '# ' >> notes && history | tail -2 | head -1 | tr -s ' ' | cu
```

To use this, just type *n* (for *notesappend*) after a command:

```
$ ./long_hard-to-remember_command --with_lots --of_flags > poorly_named_file
$ n
```

Now a *notes* file has been created (or appended to) in our cwd and we can look at it:

```
$ cat notes
# ./long_hard-to-remember_command --with_lots --of_flags > poorly_named_file
```

I use *notesappend* almost as much as I use the regular unix utilities.

clear TOP VIEW_AS_PAGE

`clear` clears your screen:

```
$ clear
```

It gives you a clean slate without erasing any of your commands, which are still preserved if you scroll up. `clear` works in some other shells, such as ipython's. In most shells—including those where `clear` doesn't work, like MySQL's and Python's—you can use *Cntrl-l* to clear the screen.

logout, exit TOP VIEW_AS_PAGE

Both `logout` and `exit` do exactly what their name suggests—quit your current session. Another way of doing the same thing is the key combination *Cntrl-D*, which works across a broad variety of different shells, including Python's and MySQL's.

sudo TOP VIEW_AS_PAGE

Based off of An Introduction to the Command-Line (on Unix-like systems) - sudo and the Root User: *sudo and the Root User* sounds like one of Aesop's Fables, but the moralist never considered file permissions and system administration a worthy subject. We saw above that, if you're unsure of your user name, the command `whoami` tells you who you're logged in as. If you list the files in the root directory:

```
$ ls -hl /
```

you'll notice that these files do not belong to you but, rather, have been created by another user named *root*. Did you know you were sharing your computer with somebody else? You're not exactly, but there is another account on your computer for the *root user* who has, as far as the computer's concerned, all the power (read Wikipedia's discussion about the superuser here). It's good that you're not the root user by default because it protects you from yourself. You don't have permission to catastrophically delete an important system file.

If you want to run a command as the superuser, you can use `sudo`. For example, you'll have trouble if you try to make a directory called *junk* in the root directory:

```
$ mkdir /junk
mkdir: cannot create directory '/junk': Permission denied
```

However, if you invoke this command as the root user, you can do it:

```
$ sudo mkdir /junk
```

provided you type in the password. Because root—not your user—owns this file, you also need `sudo` to remove this directory:

```
$ sudo rmdir /junk
```

If you want to experience life as the root user, try:

```
$ sudo -i
```

Here's what this looks like on my computer:

```
$ whoami        # check my user name
oliver
$ sudo -i       # simulate login as root
Password:
$ whoami        # check user name now
root
$ exit          # logout of root account
logout
$ whoami        # check user name again
oliver
```

Obviously, you should be cautious with `sudo`. When might using it be appropriate? The most common use case is when you have to install a program, which might want to write into a directory root owns, like `/usr/bin`, or access system files. I discuss installing new software below. You can also do terrible things with `sudo`, such as gut your whole computer with a command so unspeakable I cannot utter it in syntactically viable form. That's *sudo are em dash are eff forward slash*—it lives in infamy in an Urban Dictionary entry.

You can grant root permissions to various users by tinkering with the configuration file:

which says: *"Sudoers allows particular users to run various commands as the root user, without needing the root password."* Needless to say, only do this if you know what you're doing.

su TOP VIEW_AS_PAGE

`su` switches the user you are in the terminal. For example, to change to the user *jon*:

```
$ su jon
```

where you have to know jon's password.

wc TOP VIEW_AS_PAGE

`wc` counts the number of words, lines, or characters in a file.

Count lines:

```
$ cat myfile.txt
aaa
bbb
ccc
ddd
```

```
$ cat myfile.txt | wc -l
4
```

Count words:

```
$ echo -n joe | wc -w
1
```

Count characters:

```
$ echo -n joe | wc -c
3
```

sort TOP VIEW_AS_PAGE

From An Introduction to the Command-Line (on Unix-like systems) - sort: As you guessed, the command `sort` sorts files. It has a large man page, but we can learn its basic features by example. Let's suppose we have a file, *testsort.txt*, such that:

```
$ cat testsort.txt
vfw     34      awfjo
a       4       2
f       10      10
beb     43      c
f       2       33
f       1       ?
```

Then:

```
$ sort testsort.txt
a       4       2
beb     43      c
f       1       ?
f       10      10
f       2       33
vfw     34      awfjo
```

What happened? The default behavior of `sort` is to *dictionary sort* the rows of a file according to what's in the first column, then second column, and so on. Where the first column has the same value—*f* in this

example—the values of the second column determine the order of the rows. *Dictionary sort* means that things are sorted as they would be in a dictionary: *1,2,10* gets sorted as *1,10,2*. If you want to do a numerical sort, use the `-n` flag; if you want to sort in reverse order, use the `-r` flag. You can also sort according to a specific column. The notation for this is:

```
sort -kn,m
```

where *n* and *m* are numbers which refer to the range column *n* to column *m*. In practice, it may be easier to use a single column rather than a range so, for example:

```
sort -k2,2
```

means sort by the second column (technically from column 2 to column 2).

To sort *numerically* by the second column:

```
$ sort -k2,2n testsort.txt
f       1       ?
f       2       33
a       4       2
f       10      10
vfw     34      awfjo
beb     43      c
```

As is often the case in unix, we can combine flags as much as we like.

Question: what does this do?

```
$ sort -k1,1r -k2,2n testsort.txt
vfw     34      awfjo
f       1       ?
f       2       33
f       10      10
beb     43      c
a       4       2
```

Answer: the file has been sorted first by the first column, in reverse dictionary order, and then—where the first column is the same—by the second column in numerical order. You get the point!

Sort uniquely:

```
$ sort -u testsort.txt              # sort uniquely
```

Sort using a designated tmp directory:

```
$ sort -T /my/tmp/dir testsort.txt   # sort using a designated tmp directory
```

Behind the curtain, `sort` does its work by making temporary files, and it needs a place to put those files. By default, this is the directory set by `TMPDIR`, but if you have a giant file to sort, you might have reason to instruct `sort` to use another directory and that's what this flag does.

Sort numerically if the columns are in scientific notation:

```
$ sort -g testsort.txt
```

`sort` works particularly well with `uniq`. For example, look at the following list of numbers:

```
$ echo "2 2 2 1 2 1 3 4 5 6 6" | tr " " "\n" | sort
1
1
2
```

```
2
2
2
3
4
5
6
6
```

Find the duplicate entries:

```
$ echo "2 2 2 1 2 1 3 4 5 6 6" | tr " " "\n" | sort | uniq -d
1
2
6
```

Find the non-duplicate entries:

```
$ echo "2 2 2 1 2 1 3 4 5 6 6" | tr " " "\n" | sort | uniq -u
3
4
5
```

ssh TOP VIEW_AS_PAGE

As discussed in An Introduction to the Command-Line (on Unix-like systems) - ssh, the Secure Shell (ssh) protocol is as fundamental as tying your shoes. Suppose you want to use a computer, but it's not the computer that's in front of you. It's a different computer in some other location—say, at your university or on the Amazon cloud. `ssh` is the command that allows you to log into a computer remotely over the network. Once you've `ssh`ed into a computer, you're in its shell and can run commands on it just as if it were your personal laptop. To `ssh`, you need to know the address of the host computer you want to log into, your user name on that computer, and the password. The basic syntax is:

```
ssh username@host
```

For example:

```
$ ssh username@myhost.university.edu
```

If you have a Macintosh, you can allow users to `ssh` into your personal computer by turning `ssh` access on. Go to:

```
System Preferences > Sharing > Remote Login
```

`ssh` with the flag *-Y* to enable X11 (X Window System) forwarding:

```
$ ssh -Y username@myhost.university.edu
```

Check that this succeeded:

```
$ echo $DISPAY  # this should not be empty
$ xclock        # this should pop open X11's xclock
```

`ssh` also allows you to run a command on the remote server without logging in. For instance, to list of the contents of your remote computer's home directory, you could run:

```
$ ssh -Y username@myhost.university.edu "ls -hl"
```

Cool, eh?

The file:

```
~/.ssh/config
```

determines `ssh`'s behavior and you can create it if it doesn't exist. In the next section, you'll see how to modify this config file to use an IdentityFile, so that you're spared the annoyance of typing in your password every time you `ssh` (see also Nerderati: Simplify Your Life With an SSH Config File).

If you're frequently getting booted from a remote server after periods of inactivity, trying putting something like this into your config file:

```
ServerAliveInterval = 300
```

If this is your first encounter with `ssh`, you'd be surprised how much of the work of the world is done by `ssh`. It's worth reading the extensive man page, which gets into matters of computer security and cryptography.

ssh-keygen TOP VIEW_AS_PAGE

On your own private computer, you can `ssh` into a particular server without having to type in your password. To set this up, first generate rsa ssh keys:

```
$ mkdir -p ~/.ssh && cd ~/.ssh
$ ssh-keygen -t rsa -b 4096 -f localkey
```

This will create two files on your computer, a public key:

`~/.ssh/localkey.pub`

and a private key:

`~/.ssh/localkey`

You can share your public key, but *do not give anyone your private key!* Suppose you want to `ssh` into *myserver.com*. Normally, that's:

```
$ ssh myusername@myserver.com
```

Add these lines to your *~/.ssh/config* file:

```
Host Myserver
    HostName myserver.com
    User myusername
    IdentityFile ~/.ssh/localkey
```

Now `cat` your public key and paste it into:

`~/.ssh/authorized_keys`

on the remote machine (i.e., on *myserver.com*). Now on your local computer, you can `ssh` without a password:

```
$ ssh Myserver
```

You can also use this technique to push to github.com, without having to punch your password in each time, as described here.

scp TOP VIEW_AS_PAGE

If you have `ssh` access to a remote computer and want to copy its files to your local computer, you can use `scp` according to the syntax:

```
scp username@host:/some/path/on/remote/machine /some/path/on/my/machine
```

However, I would advise using `rsync` instead of `scp`: it does the same thing only better.

rsync TOP VIEW_AS_PAGE

`rsync` can remotely sync files to or from a computer to which you have `ssh` access. The basic syntax is:

```
rsync source destination
```

For example, copy files from a remote machine:

```
$ rsync username@host:/some/path/on/remote/machine /some/path/on/my/machine/
```

Copy files to a remote machine:

```
$ rsync /some/path/on/my/machine username@host:/some/path/on/remote/machine/
```

You can also use it to sync two directories on your local machine:

```
$ rsync directory1 directory2
```

The great thing about `rsync` is that, if it fails or you stop it in the middle, you can re-start it and it will pick up where it left off. It does not blindly copy directories (or files) but rather syncronizes them—which is to say, makes them the same. If, for example, you try to copy a directory that you already have, `rsync` will detect this and transfer no files.

I like to use the flags:

```
$ rsync -azv --progress username@myhost.university.edu:/my/source /my/destination/
```

The *-a* flag is for *archive*, which *"ensures that symbolic links, devices, attributes, permissions, ownerships, etc. are preserved in the transfer"*; the *-z* flag compresses files during the transfer; *-v* is for *verbose*; and *--progress* shows you your progress. I've enshrined this in an alias:

```
alias yy="rsync -azv --progress"
```

Preview `rsync`'s behavior without actually transferring any files:

```
$ rsync --dry-run username@myhost.university.edu:/my/source /my/destination/
```

Do not copy certain directories from the source:

```
$ rsync --exclude mydir username@myhost.university.edu:/my/source /my/destination/
```

In this example, the directory */my/source/mydir* will not be copied, and you can omit more directories by repeating the *--exclude* flag.

Copy the files pointed to by the symbolic links (*"transform symlink into referent file/dir"*) with the *--L* flag:

```
$ rsync -azv -L --progress username@myhost.university.edu:/my/source /my/destination/
```

Note: using a trailing slash can alter the behavior of `rsync`. For example:

```
$ mkdir folder1 folder2 folder3 # make directories
$ touch folder1/{file1,file2}   # create empty files
$ rsync -av folder1 folder2     # no trailing slash
$ rsync -av folder1/ folder3    # trailing slash
```

No trailing slash copies the directory *folder1* into *folder2*:

```
$ tree folder2
folder2
└── folder1
    ├── file1
    └── file2
```

whereas using a trailing slash copies the files in *folder1* into *folder2*:

```
$ tree folder3
folder3
```

```
├── file1
└── file2
```

source, export TOP VIEW_AS_PAGE

From An Introduction to the Command-Line (on Unix-like systems) - Source and Export: Question: if we create some variables in a script and exit, what happens to those variables? Do they disappear? The answer is, yes, they do. Let's make a script called *test_src.sh* such that:

```
$ cat ./test_src.sh
#!/bin/bash

myvariable=54
echo $myvariable
```

If we run it and then check what happened to the variable on our command line, we get:

```
$ ./test_src.sh
54
$ echo $myvariable


```

The variable is undefined. The command `source` is for solving this problem. If we want the variable to persist, we run:

```
$ source ./test_src.sh
54
$ echo $myvariable
54
```

and—voilà!—our variable exists in the shell. An equivalent syntax for sourcing uses a dot:

```
$ . ./test_src.sh  # this is the same as "source ./test_src.sh"
54
```

But now observe the following. We'll make a new script, *test_src_2.sh*, such that:

```
$ cat ./test_src_2.sh
#!/bin/bash

echo $myvariable
```

This script is also looking for *$myvariable*. Running it, we get:

```
$ ./test_src_2.sh


```

Nothing! So *$myvariable* is defined in the shell but, if we run another script, its existence is unknown. Let's amend our original script to add in an `export`:

```
$ cat ./test_src.sh
#!/bin/bash

export myvariable=54  # export this variable
echo $myvariable
```

Now what happens?

```
$ ./test_src.sh
54
$ ./test_src_2.sh
```

Still nothing! Why? Because we didn't `source` *test_src.sh*. Trying again:

```
$ source ./test_src.sh
54
$ ./test_src_2.sh
54
```

So, at last, we see how to do this. If we want access on the shell to a variable which is defined inside a script, we must `source` that script. If we want *other* scripts to have access to that variable, we must `source` plus `export`.

In TOP VIEW_AS_PAGE

From An Introduction to the Command-Line (on Unix-like systems) - Links: If you've ever used the *Make Alias* command on a Macintosh (not to be confused with the unix command `alias`), you've already developed intuition for what a link is. Suppose you have a file in one folder and you want that file to exist in another folder simultaneously. You could copy the file, but that would be wasteful. Moreover, if the file changes, you'll have to re-copy it—a huge ball-ache. Links solve this problem. A link to a file is a stand-in for the original file, often used to access the original file from an alternate file path. It's not a copy of the file but, rather, points to the file.

To make a symbolic link, use the command `ln`:

```
$ ln -s /path/to/target/file mylink
```

This produces:

```
mylink --> /path/to/target/file
```

in the cwd, as `ls -hl` will show. Note that removing *mylink*:

```
$ rm mylink
```

does not affect our original file.

If we give the target (or source) path as the sole argument to `ln`, the name of the link will be the same as the source file's. So:

```
$ ln -s /path/to/target/file
```

produces:

```
file --> /path/to/target/file
```

Links are incredibly useful for all sorts of reasons—the primary one being, as we've already remarked, if you want a file to exist in multiple locations without having to make extraneous, space-consuming copies. You can make links to directories as well as files. Suppose you add a directory to your `PATH` that has a particular version of a program in it. If you install a newer version, you'll need to change the `PATH` to include the new directory. However, if you add a link to your `PATH` and keep the link always pointing to the most up-to-date directory, you won't need to keep fiddling with your `PATH`. The scenario could look like this:

```
$ ls -hl myprogram
current -> version3
version1
```

```
version2
version3
```

(where I'm hiding some of the output in the long listing format.) In contrast to our other examples, the link is in the same directory as the target. Its purpose is to tell us which version, among the many crowding a directory, we should use.

Another good practice is putting links in your home directory to folders you often use. This way, navigating to those folders is easy when you log in. If you make the link:

```
~/MYLINK --> /some/long/and/complicated/path/to/an/often/used/directory
```

then you need only type:

```
$ cd MYLINK
```

rather than:

```
$ cd /some/long/and/complicated/path/to/an/often/used/directory
```

Links are everywhere, so be glad you've made their acquaintance!

readlink TOP VIEW_AS_PAGE

`readlink` provides a way to get the absolute path of a directory.

Get the absolute path of the directory *mydir*:

```
$ readlink -m mydir
```

Get the absolute path of the cwd:

```
$ readlink -m .
```

Note this is different than:

```
$ pwd
```

because I'm using the term *absolute path* to express not only that the path is not a relative path, but also to denote that it's free of symbolic links. So if we have the link discussed above:

```
~/MYLINK --> /some/long/and/complicated/path/to/an/often/used/directory
```

then:

```
$ cd ~/MYLINK
$ readlink -m .
/some/long/and/complicated/path/to/an/often/used/directory
```

One profound annoyance: `readlink` doesn't work the same way on Mac unix (Darwin). To get the proper one, you need to download the GNU coreutils.

git TOP VIEW_AS_PAGE

`git` is a famous utility for version control in software development written by Linus Torvalds. Everyone doing serious software development uses version control, which is related to the familiar concept of saving a file. Suppose you're writing an article or a computer program. As you progress, you save the file at different checkpoints ("*file_v1*", "*file_v2*", "*file_v3*"). `git` is an extension of this idea except, instead of saving a single file, it can take a snapshot of the state of all the files and folders in your project directory. In `git` jargon, this snapshot is called a *"commit"* and, forever after, you can revisit the state of your project at this snapshot.

The popular site Github took the utility `git` and added servers in the cloud which (to a first approximation) respond only to `git` commands. This allows people to both back up their code base remotely and easily collaborate on software development.

`git` is such a large subject (the utility has many sub-commands) I've given it a full post here.

sleep TOP VIEW_AS_PAGE

The command `sleep` pauses for an amount of time specified in seconds. For example, sleep for 5 seconds:

```
$ sleep 5
```

ps, pstree, jobs, bg, fg, kill, top, htop TOP VIEW_AS_PAGE

Borrowing from An Introduction to the Command-Line (on Unix-like systems) - Processes and Running Processes in the Background: Processes are a deep subject intertwined with topics like memory usage, threads, scheduling, and parallelization. My understanding of this stuff—which encroaches on the domain of operating systems—is narrow, but here are the basics from a unix-eye-view. The command `ps` displays information about your processes:

```
$ ps      # show process info
$ ps -f   # show verbose process info
```

What if you have a program that takes some time to run? You run the program on the command line but, while it's running, your hands are tied. You can't do any more computing until it's finished. Or can you? You can—by running your program in the background. Let's look at the command `sleep` which pauses for an amount of time specified in seconds. To run things in the background, use an ampersand:

```
$ sleep 60 &
[1] 6846
$ sleep 30 &
[2] 6847
```

The numbers printed are the *PIDs* or *process IDs*, which are unique identifiers for every process running on our computer. Look at our processes:

```
$ ps -f
  UID   PID  PPID   C STIME   TTY           TIME CMD
  501  5166  5165   0 Wed12PM ttys008    0:00.20 -bash
  501  6846  5166   0 10:57PM ttys008    0:00.01 sleep 60
  501  6847  5166   0 10:57PM ttys008    0:00.00 sleep 30
```

The header is almost self-explanatory, but what's *TTY*? Wikipedia tells us that this is an anachronistic name for the terminal that derives from *TeleTYpewriter*. Observe that bash itself is a process which is running. We also see that this particular bash—PID of 5166—is the *parent* process of both of our `sleep` commands (*PPID* stands for *parent process ID*). Like directories, processes are hierarchical. Every directory, except for the root directory, has a parent directory and so every process, except for the first process—*init* in unix or *launchd* on the Mac—has a parent process. Just as `tree` shows us the directory hierarchy, `pstree` shows us the process hierarchy:

```
$ pstree
-+= 00001 root /sbin/launchd
 |-+= 00132 oliver /sbin/launchd
 | |-+= 00252 oliver /Applications/Utilities/Terminal.app/Contents/MacOS/Terminal
 | | \-+= 05165 root login -pf oliver
 | |   \-+= 05166 oliver -bash
 | |     |-- = 06846 oliver sleep 30
 | |     |-- = 06847 oliver sleep 60
 | |     \-+= 06848 oliver pstree
 | |       \--- 06849 root ps -axwwo user,pid,ppid,pgid,command
```

This command actually shows every single process on the computer, but I've cheated a bit and cut out everything but the processes we're looking at. We can see the parent-child relationships: `sleep` and `pstree` are children of bash, which is a child of login, which is a child of Terminal, and so on. With vanilla `ps`, every process we saw began in a terminal—that's why it had a TTY ID. However, we're probably running Safari or Chrome. How do we see these processes as well as the myriad others running on our system? To see all processes, not just those spawned by terminals, use the *-A* flag:

```
$ ps -A   # show all process info
$ ps -fA  # show all process info (verbose)
```

Returning to our example with `sleep`, we can see the jobs we're currently running with the command `jobs`:

```
$ jobs
[1]-  Running                 sleep 60 &
[2]+  Running                 sleep 30 &
```

What if we run our `sleep` job in the foreground, but it's holding things up and we want to move it to the background? *Cntrl-z* will pause or stop a job and the commands `bg` and `fg` set it running in either the background or the foreground.

```
$ sleep 60  # pause job with control-z
^Z
[1]+  Stopped                 sleep 60
$ jobs      # see jobs
[1]+  Stopped                 sleep 60
$ bg        # set running in background
[1]+ sleep 60 &
$ jobs      # see jobs
[1]+  Running                 sleep 60 &
$ fg        # bring job from background to foreground
sleep 60
```

We can also `kill` a job:

```
$ sleep 60 &
[1] 7161
$ jobs
[1]+  Running                 sleep 60 &
$ kill %1
$
[1]+  Terminated: 15          sleep 60
```

In this notation `%n` refers to the *n*th job. Recall we can also kill a job in the foreground with *Cntrl-c*. More generally, if we didn't submit the command ourselves, we can kill any process on our computer using the PID. Suppose we want to kill the terminal in which we are working. Let's `grep` for it:

```
$ ps -Af | grep Terminal
  501   252   132   0 11:02PM ??         0:01.66 /Applications/Utilities/Termi
  501  1653  1532   0 12:09AM ttys000    0:00.00 grep Terminal
```

We see that this particular process happens to have a PID of 252. `grep` returned any process with the word *Terminal*, including the `grep` itself. We can be more precise with `awk` and see the header, too:

```
$ ps -Af | awk 'NR==1 || $2==252'
  UID   PID  PPID   C STIME   TTY           TIME CMD
  501   252   132   0 11:02PM ??         0:01.89 /Applications/Utilities/Termi
```

(that's print the first row OR anything with the 2nd field equal to 252.) Now let's kill it:

```
$ kill 252
```

poof!—our terminal is gone.

Running stuff in the background is useful, especially if you have a time-consuming program. If you're scripting a lot, you'll often find yourself running something like this:

```
$ script.py > out.o 2> out.e &
```

i.e., running something in the background and saving both the output and error.

Two other commands that come to mind are `time`, which times how long your script takes to run, and `nohup` (*"no hang up"*), which allows your script to run even if you quit the terminal:

```
$ time script.py > out.o 2> out.e &
$ nohup script.py > out.o 2> out.e &
```

As we mentioned above, you can also set multiple jobs to run in the background, in parallel, from a loop:

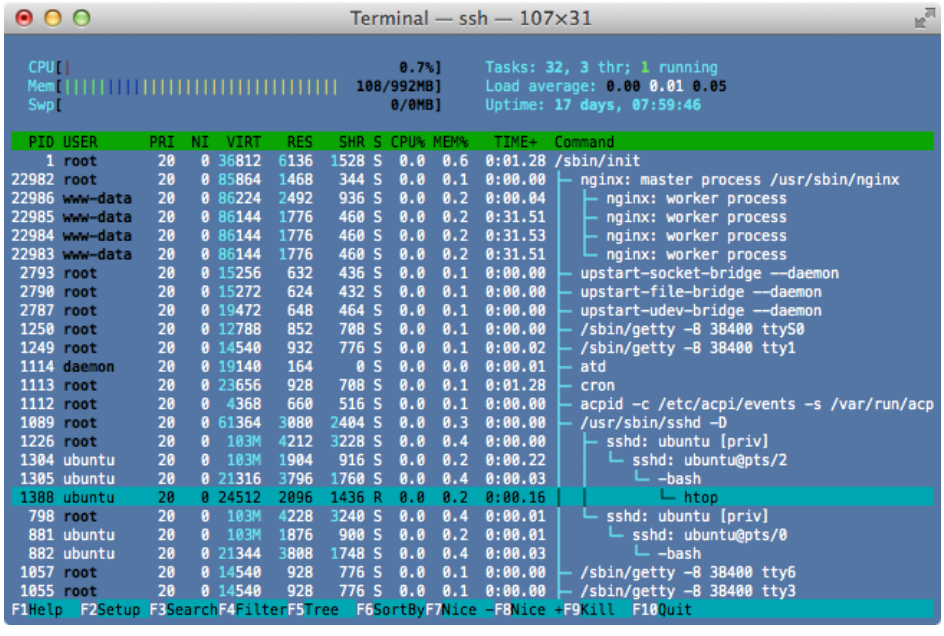
```
$ for i in 1 2 3; do { echo "***"$i; sleep 60 & } done
```

(Of course, you'd want to run something more useful than `sleep`!) If you're lucky enough to work on a large computer cluster shared by many users—some of whom may be running memory- and time-intensive programs—then scheduling different users' jobs is a daily fact of life. At work we use a queuing system called the Sun Grid Engine to solve this problem. I wrote a short SGE wiki here.

To get a dynamic view of your processes, loads of other information, and sort them in different ways, use `top`:

```
$ top
```

Here's a screenshot of `htop`—a souped-up version of `top`—running on Ubuntu:



`htop` can show us a traditional `top` output split-screened with a process tree. We see various users—*root*, *ubuntu*, and *www-data*—and we see various processes, including *init* which has a PID of 1. `htop` also shows us the percent usage of each of our cores—here there's only one and we're using just 0.7%. Can you guess what this computer might be doing? I'm using it to host a website, as the process for nginx, a popular web server, gives away.

nohup TOP VIEW_AS_PAGE

As we mentioned above, `nohup` (*"no hang up"*) allows your script to run even if you quit the terminal, as in:

```
$ nohup script.py > out.o 2> out.e &
```

If you start such a process, you can squelch it by using `kill` on its process ID or, less practically, by shutting down your computer. You can track down your process with:

```
$ ps -Af | grep script.py
```

`nohup` is an excellent way to make a job running in the background more robust.

time TOP VIEW_AS_PAGE

As we mentioned above, `time` tells you how long your script or command took to run. This is useful for benchmarking the efficency of your program, among other things. For example, to time `sleep`:

```
$ time sleep 10
real    0m10.003s
user    0m0.000s
sys     0m0.000s
```

How do you tell how much memory a program or command consumed? Contrary to what its name suggests, the `time` command also gives us this information with the *-v* flag:

```
$ time -v sleep 10
-bash: -v: command not found
```

What?! We got an error. Why?

```
$ type time
time is a shell keyword
```

Confusingly, the reason is that *time* is actually a keyword, so we can't invoke it with flags as usual. To use it, we have to call the program by its full path:

```
$ /usr/bin/time -v sleep 10
        Command being timed: "sleep 10"
        User time (seconds): 0.00
        System time (seconds): 0.00
        Percent of CPU this job got: 0%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:10.00
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 2560
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 202
        Voluntary context switches: 2
        Involuntary context switches: 0
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

Now we get the information payload.

seq TOP VIEW_AS_PAGE

`seq` prints a sequence of numbers.

Display numbers 1 through 5:

```
$ seq 1 5
1
2
3
4
5
```

You can achieve the same result with:

```
$ echo {1..5} | tr " " "\n"
1
2
3
4
5
```

If you add a number in the middle of your `seq` range, this will be the "step":

```
$ seq 1 2 10
1
3
5
7
9
```

cut TOP VIEW_AS_PAGE

`cut` cuts one or more columns from a file, and delimits on tab by default. Suppose a file, *sample.blast.txt*, is:

```
TCONS_00007936|m.162    gi|27151736|ref|NP_006727.2|    100.00  324
TCONS_00007944|m.1236   gi|55749932|ref|NP_001918.3|    99.36   470
TCONS_00007947|m.1326   gi|157785645|ref|NP_005867.3|   91.12   833
TCONS_00007948|m.1358   gi|157785645|ref|NP_005867.3|   91.12   833
```

Then:

```
$ cat sample.blast.txt | cut -f2
gi|27151736|ref|NP_006727.2|
gi|55749932|ref|NP_001918.3|
gi|157785645|ref|NP_005867.3|
gi|157785645|ref|NP_005867.3|
```

You can specify the delimiter with the *-d* flag, so:

```
$ cat sample.blast.txt | cut -f2 | cut -f4 -d"|"
NP_006727.2
NP_001918.3
NP_005867.3
NP_005867.3
```

although this is long-winded and in this case we can achieve the same result simply with:

```
$ cat sample.blast.txt | cut -f5 -d"|"
NP_006727.2
NP_001918.3
NP_005867.3
NP_005867.3
```

Don't confuse `cut` with its non-unix namesake on Macintosh, which deletes text while copying it to the clipboard.

Tip: If you're a Vim user, running `cut` as a system command within Vim is a neat way to filter text. Read more: Wiki Vim - System Commands in Vim.

paste TOP VIEW_AS_PAGE

`paste` joins files together in a column-wise fashion. Another way to think about this is in contrast to `cat`, which joins files vertically. For example:

```
$ cat file1.txt
a
b
c
```

```
$ cat file2.txt
1
2
3
```

```
$ paste file1.txt file2.txt
a       1
b       2
c       3
```

Paste with a delimiter:

```
$ paste -d";" file1.txt file2.txt
a;1
b;2
c;3
```

As with `cut`, `paste`'s non-unix namesake on Macintosh—printing text from the clipboard—is a different beast entirely.

A neat feature of `paste` is the ability to put different rows of a file on the same line. For example, if the file *sample.fa* is:

```
>TCONS_00046782
FLLRQNDFHSVTQAGVQWCDLGSLQSLPPRLKQISCLSLLSSWDYRHRPPHPAFFLFFFLF
>TCONS_00046782
MRWHMPIIPALWEAEVSGSPDVRSLRPTWPTTPSLLKTKNKTKQNISWAWCMCL
>TCONS_00046782
MFCFVLFFVFSRDGVVGQVGLKLLTSGDPLTSASQSAGIIGMCHRIQPWLLIY
```

(This is fasta format from bioinformatics). Then:

```
$ cat sample.fa | paste - -
>TCONS_00046782 FLLRQNDFHSVTQAGVQWCDLGSLQSLPPRLKQISCLSLLSSWDYRHRPPHPAFFLFFFLF
>TCONS_00046782 MRWHMPIIPALWEAEVSGSPDVRSLRPTWPTTPSLLKTKNKTKQNISWAWCMCL
>TCONS_00046782 MFCFVLFFVFSRDGVVGQVGLKLLTSGDPLTSASQSAGIIGMCHRIQPWLLIY
```

And you can do this with as many dashes as you like.

awk TOP VIEW_AS_PAGE

From An Introduction to the Command-Line (on Unix-like systems) - awk: `awk` and `sed` are command line utilities which are themselves programming languages built for text processing. As such, they're vast subjects as these huge manuals—GNU Awk Guide, Bruce Barnett's Awk Guide, GNU Sed Guide—attest. Both of these languages are almost antiques which have been pushed into obsolescence by Perl and Python. For anything serious, you probably don't want to use them. However, their syntax makes them useful for simple parsing or text manipulation problems that crop up on the command line. Writing a simple line of

awk can be faster and less hassle than hauling out Perl or Python.

The key point about awk is, it works line by line. A typical awk construction is:

```
cat file.txt | awk '{ some code }'
```

Awk executes its code once every line. Let's say we have a file, *test.txt*, such that:

```
$ cat test.txt
1       c
3       c
2       t
1       c
```

In awk, the notation for the first field is *$1*, *$2* is for second, and so on. The whole line is *$0*. For example:

```
$ cat test.txt | awk '{print}'     # print full line
1       c
3       c
2       t
1       c
```

```
$ cat test.txt | awk '{print $0}'  # print full line
1       c
3       c
2       t
1       c
```

```
$ cat test.txt | awk '{print $1}'  # print col 1
1
3
2
1
```

```
$ cat test.txt | awk '{print $2}'  # print col 2
c
c
t
c
```

There are two exceptions to the execute code per line rule: anything in a *BEGIN* block gets executed before the file is read and anything in an *END* block gets executed after it's read. If you define variables in awk they're global and persist rather than being cleared every line. For example, we can concatenate the elements of the first column with an @ delimiter using the variable *x*:

```
$ cat test.txt | awk 'BEGIN{x=""}{x=x"@"$1; print x}'
@1
@1@3
@1@3@2
@1@3@2@1
```

```
$ cat test.txt | awk 'BEGIN{x=""}{x=x"@"$1}END{print x}'
@1@3@2@1
```

Or we can sum up all values in the first column:

```
$ cat test.txt | awk '{x+=$1}END{print x}'  # x+=$1 is the same as x=x+$1
7
```

Awk has a bunch of built-in variables which are handy: *NR* is the row number; *NF* is the total number of fields; and *OFS* is the output delimiter. There are many more you can read about here. Continuing with our

very contrived examples, let's see how these can help us:

```
$ cat test.txt | awk '{print $1"\t"$2}'        # write tab explicitly
1       c
3       c
2       t
1       c
```

```
$ cat test.txt | awk '{OFS="\t"; print $1,$2}' # set output field separator to
1       c
3       c
2       t
1       c
```

Setting *OFS* spares us having to type a *"\t"* every time we want to print a tab. We can just use a comma instead. Look at the following three examples:

```
$ cat test.txt | awk '{OFS="\t"; print $1,$2}'        # print file as is
1       c
3       c
2       t
1       c
```

```
$ cat test.txt | awk '{OFS="\t"; print NR,$1,$2}'     # print row num
1       1       c
2       3       c
3       2       t
4       1       c
```

```
$ cat test.txt | awk '{OFS="\t"; print NR,NF,$1,$2}'  # print row & field num
1       2       1       c
2       2       3       c
3       2       2       t
4       2       1       c
```

So the first command prints the file as it is. The second command prints the file with the row number added in front. And the third prints the file with the row number in the first column and the number of fields in the second—in our case always two. Although these are purely pedagogical examples, these variables can do a lot for you. For example, if you wanted to print the 3rd row of your file, you could use:

```
$ cat test.txt | awk '{if (NR==3) {print $0}}' # print the 3rd row of your fil
2       t
```

```
$ cat test.txt | awk '{if (NR==3) {print}}'    # same thing, more compact synt
2       t
```

```
$ cat test.txt | awk 'NR==3'                   # same thing, most compact synt
2       t
```

Sometimes you have a file and you want to check if every row has the same number of columns. Then use:

```
$ cat test.txt | awk '{print NF}' | sort -u
2
```

In awk *$NF* refers to the *contents* of the last field:

```
$ cat test.txt | awk '{print $NF}'
c
c
t
c
```

An important point is that *by default awk delimits on white-space*, not tabs (unlike, say, `cut`). White space means any combination of spaces and tabs. You can tell awk to delimit on anything you like by using the *-F* flag. For instance, let's look at the following situation:

```
$ echo "a b" | awk '{print $1}'
a
```

```
$ echo "a b" | awk -F"\t" '{print $1}'
a b
```

When we feed *a space b* into awk, *$1* refers to the first field, *a*. However, if we explicitly tell awk to delimit on tabs, then *$1* refers to *a b* because it occurs before a tab.

You can also use shell variables inside your awk by importing them with the *-v* flag:

```
$ x=hello
$ cat test.txt | awk -v var=$x '{ print var"\t"$0 }'
hello   1       c
hello   3       c
hello   2       t
hello   1       c
```

And you can write to multiple files from inside awk:

```
$ cat test.txt | awk '{if ($1==1) {print > "file1.txt"} else {print > "file2.txt"}}'
```

```
$ cat file1.txt
1       c
1       c
```

```
$ cat file2.txt
3       c
2       t
```

For loops in awk:

```
$ echo joe | awk '{for (i = 1; i <= 5; i++) {print i}}'
1
2
3
4
5
```

Question: In the following case, how would you print the row numbers such that the first field equals the second field?

```
$ echo -e "a\ta\na\tc\na\tz\na\ta"
a       a
a       c
a       z
a       a
```

Here's the answer:

```
$ echo -e "a\ta\na\tc\na\tz\na\ta" | awk '$1==$2{print NR}'
1
4
```

Question: How would you print the average of the first column in a text file?

```
$ cat file.txt | awk 'BEGIN{x=0}{x=x+$1;}END{print x/NR}'
```

NR is a special variable representing row number.

The take-home lesson is, you can do tons with awk, but you don't want to do too much. Anything that you can do crisply on one, or a few, lines is awk-able. For more involved scripting examples, see An Introduction to the Command-Line (on Unix-like systems) - More awk examples.

sed TOP VIEW_AS_PAGE

From An Introduction to the Command-Line (on Unix-like systems) - sed: Sed, like awk, is a full-fledged language that is convenient to use in a very limited sphere (GNU Sed Guide). I mainly use it for two things: *(1)* replacing text, and *(2)* deleting lines. Sed is often mentioned in the same breath as *regular expressions* although, like the rest of the world, I'd use Perl and Python when it comes to that. Nevertheless, let's see what sed can do.

Sometimes the first line of a text file is a header and you want to remove it. Then:

```
$ cat test_header.txt
This is a header
1       asdf
2       asdf
2       asdf
```

```
$ cat test_header.txt | sed '1d'    # delete the first line
1       asdf
2       asdf
2       asdf
```

To remove the first 3 lines:

```
$ cat test_header.txt | sed '1,3d'  # delete lines 1-3
2       asdf
```

1,3 is sed's notation for the range 1 to 3. We can't do much more without entering regular expression territory. One sed construction is:

```
/pattern/d
```

where *d* stands for delete if the pattern is matched. So to remove lines beginning with #:

```
$ cat test_comment.txt
1       asdf
# This is a comment
2       asdf
# This is a comment
2       asdf
```

```
$ cat test_comment.txt | sed '/^#/d'
1       asdf
2       asdf
2       asdf
```

Another construction is:

```
s/A/B/
```

where *s* stands for substitute. So this means replace *A* with *B*. By default, this only works for the first occurrence of *A*, but if you put a *g* at the end, for *group*, all *A*s are replaced:

```
s/A/B/g
```

For example:

```
$ # replace 1st occurrence of kitty with X
$ echo "hello kitty. goodbye kitty" | sed 's/kitty/X/'
hello X. goodbye kitty
```

```
$ # same thing. using | as a separator is ok
$ echo "hello kitty. goodbye kitty" | sed 's|kitty|X|'
hello X. goodbye kitty
```

```
$ # replace all occurrences of kitty
$ echo "hello kitty. goodbye kitty" | sed 's/kitty/X/g'
hello X. goodbye X
```

This is such a useful ability that all text editors allow you to perform *find-and-replace* as well. By replacing some text with nothing, you can also use this as a delete:

```
$ echo "hello kitty. goodbye kitty" | sed 's|kitty||'
hello . goodbye kitty
```

```
$ echo "hello kitty. goodbye kitty" | sed 's|kitty||g'
hello . goodbye
```

Sed is especially good when you're trying to rename batches of files on the command line. I often have occasion to use it in for loops:

```
$ touch file1.txt file2.txt file3.txt
$ for i in file*; do echo $i; j=$( echo $i | sed 's|.txt|.html|'); mv $i $j; d
file1.txt
file2.txt
file3.txt
$ ls
file1.html  file2.html  file3.html
```

Sed has the ability to edit files in place with the *-i* flag—that is to say, modify the file wihout going through the trouble of creating a new file and doing the re-naming dance. For example, to add the line *This is a header* to the top of *myfile.txt*:

```
$ sed -i '1i This is a header' myfile.txt
```

Delete the first line—e.g., a header—from a file:

```
$ sed -i '1d' myfile.txt
```

date, cal TOP VIEW_AS_PAGE

`date` prints the date:

```
$ date
Sat Mar 21 18:23:56 EDT 2014
```

You can choose among many formats. For example, if the date were March 10, 2012, then:

```
$ date "+%y%m%d"
120310
```

```
$ date "+%D"
03/10/12
```

Print the date in seconds since 1970:

```
$ date "+%s"
```

`cal` prints a calendar for the current month, as in:

```
$ cal

     January
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

See the calendar for the whole year:

```
$ cal -y
```

See the calendar for, say, December 2011:

```
$ cal 12 2011
```

gzip, gunzip, bzip2, bunzip2 TOP VIEW_AS_PAGE

It's a good practice to conserve disk space whenever possible and unix has many file compression utilities for this purpose, `gzip` and `bzip2` among them. File compression is a whole science unto itself which you can read about here.

Zip a file:

```
$ gzip file
```

(The original file disappears and only the .gz file remains)

Unzip:

```
$ gunzip file.gz
```

Bunzip:

```
$ bunzip2 file.bz2
```

You can pipe into these commands!

```
$ cat file.gz | gunzip | head
```

```
$ cat file | awk '{print $1"\t"100}' | gzip > file2.gz
```

The first preserves the zipped file while allowing you to look at it, while the second illustrates how one may save space by creating a zipped file right off the bat (in this case, with some random `awk`).

`cat` a file without a unzipping it:

```
$ zcat file.gz
```

`less` a file without a unzipping it:

```
$ zless file.gz
```

To emphasize the point again, if you're dealing with large data files, you should always compress them to save space.

tar TOP VIEW_AS_PAGE

`tar` rolls, or glues, an entire directory structure into a single file.

Tar a directory named *dir* into a tarball called *dir.tar*:

```
$ tar -cvf dir.tar dir
```

(The original dir remains) The options I'm using are *-c* for *"create a new archive containing the specified items"*; *-f* for *"write the archive to the specified file"*; and *-v* for verbose.

To untar, use the *-x* flag, which stands for *extract*:

```
$ tar -xvf dir.tar
```

Tar and zip a directory *dir* into a zipped tarball *dir.tar.gz*:

```
$ tar -zcvf dir.tar.gz dir
```

Extract plus unzip:

```
$ tar -zxvf dir.tar.gz
```

uniq TOP VIEW_AS_PAGE

`uniq` filters a file leaving only the unique lines, provided the file is sorted. Suppose:

```
$ cat test.txt
aaaa
bbbb
aaaa
aaaa
cccc
cccc
```

Then:

```
$ cat test.txt | uniq
aaaa
bbbb
aaaa
cccc
```

This can be thought of as a local uniquing—adjacent rows are not the same, but you can still have a repeated row in the file. If you want the global unique, sort first:

```
$ cat test.txt | sort | uniq
aaaa
bbbb
cccc
```

This is identical to:

```
$ cat test.txt | sort -u
aaaa
bbbb
cccc
```

`uniq` also has the ability to show you only the lines that are *not* unique with the *duplicate* flag:

```
$ cat test.txt | sort | uniq -d
aaaa
cccc
```

And `uniq` can count the number of distinct rows in a file (provided it's sorted):

```
cat test.txt | sort | uniq -c
      3 aaaa
      1 bbbb
      2 cccc
```

dirname, basename TOP VIEW_AS_PAGE

`dirname` and `basename` grab parts of a file path:

```
$ basename /some/path/to/file.txt
file.txt
```

```
$ dirname /some/path/to/file.txt
/some/path/to
```

The first gets the file name, the second the directory in which the file resides. To say the same thing a different way, `dirname` gets the directory minus the file, while `basename` gets the file minus the directory.

You can play with these:

```
$ ls $( dirname $( which my_program ) )
```

This would list the files wherever *my_program* lives.

In a bash script, it's sometimes useful to grab the directory where the script itself resides and store this path in a variable:

```
# get the directory in which your script itself resides
d=$( dirname $( readlink -m $0 ) )
```

set, unset TOP VIEW_AS_PAGE

Use `set` to set various properties of your shell, somewhat analogous to a Preferences section in a GUI.

E.g., use vi style key bindings in the terminal:

```
$ set -o vi
```

Use emacs style key bindings in the terminal:

```
$ set -o emacs
```

You can use `set` with the *-x* flag to debug:

```
set -x  # activate debugging from here
.
.
```

```
.
set +x  # de-activate debugging from here
```

This causes all commands to be echoed to std:err before they are run. For example, consider the following script:

```
#!/bin/bash

set -eux

# the other flags are:
# -e Exit immediately if a simple command exits with a non-zero status
# -u Treat unset variables as an error when performing parameter expansion

echo hello
sleep 5
echo joe
```

This will echo every command before running it. The output is:

```
+ echo hello
hello
+ sleep 5
+ echo joe
joe
```

Changing gears, you can use `unset` to clear variables, as in:

```
$ TEST=asdf
$ echo $TEST
asdf
$ unset TEST
$ echo $TEST
```

env TOP VIEW_AS_PAGE

There are 3 notable things about `env`. First, if you run it as standalone, it will print out all the variables and functions set in your environment:

```
$ env
```

Second, as discussed in An Introduction to the Command-Line (on Unix-like systems) - The Shebang, you can use `env` to avoid hard-wired paths in your shebang. Compare this:

```
#!/usr/bin/env python
```

to this:

```
#!/some/path/python
```

The script with the former shebang will conveniently be interpreted by whichever `python` is first in your `PATH`; the latter will be interpreted by */some/path/python*.

And, third, as Wikipedia says, `env` can run a utility *"in an altered environment without having to modify the currently existing environment."* I never have occasion to use it this way but, since it was in the news recently, look at this example (stolen from here):

```
$ env 'COLOR=RED' bash -c 'echo "My favorite color is $COLOR"'
My favorite color is RED
```

```
$ echo $COLOR
```

The *COLOR* variable is only defined temporarily for the purposes of the `env` statement (when we `echo` it afterwards, it's empty). This construction sets us up to understand the bash shellshock bug, which Stack Exchange illustrates using `env`:

```
$ env x='() { :;}; echo vulnerable' bash -c "echo this is a test"
```

I unpack this statement in a blog post.

uname TOP VIEW_AS_PAGE

As its man page says, `uname` prints out various system information. In the simplest form:

```
$ uname
Linux
```

If you use the *-a* flag for *all*, you get all sorts of information:

```
$ uname -a
Linux my.system 2.6.32-431.11.2.el6.x86_64 #1 SMP Tue Mar 25 19:59:55 \
UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
```

df, du TOP VIEW_AS_PAGE

`df` reports *"file system disk space usage"*. This is the command you use to see how many much space you have left on your hard disk. The *-h* flag means "human readable," as usual. To see the space consumption on my mac, for instance:

```
$ df -h .
Filesystem      Size  Used Avail Use% Mounted on
/dev/disk0s3    596G  440G  157G  74% /
```

`df` shows you all of your mounted file systems. For example, if you're familiar with Amazon Web Services (AWS) jargon, `df` is the command you use to examine your mounted EBS volumes. They refer to it in the "Add a Volume to Your Instance" tutorial.

`du` is similar to `df` but it's for checking the sizes of individual directories. E.g.:

```
$ du -sh myfolder
284M    myfolder
```

If you wanted to check how much space each folder is using in your `HOME` directory, you could do:

```
$ cd
$ du -sh *
```

This will probably take a while. Also note that there's some rounding in the calculation of how much space folders and files occupy, so the numbers `df` and `du` return may not be exact.

Find all the files in the directory */my/dir* in the gigabyte range:

```
$ du -sh /my/dir/* | awk '$1 ~ /G/'
```

bind TOP VIEW_AS_PAGE

As discussed in detail in An Introduction to the Command-Line (on Unix-like systems) - Working Faster with Readline Functions and Key Bindings, *ReadLine Functions* (GNU Documentation) allow you to take all sorts of shortcuts in the terminal. You can see all the Readline Functions by entering:

```
$ bind -P  # show all Readline Functions and their key bindings
$ bind -l  # show all Readline Functions
```

Four of the most excellent Readline Functions are:
forward-word - jump cursor forward a word
backward-word - jump cursor backward a word
history-search-backward - scroll through your bash history backward
history-search-forward - scroll through your bash history forward
For the first two, you can use the default Emacs way:
Meta-f - jump forward one word
Meta-b - jump backward one word
However, reaching for the *Esc* key is a royal pain in the ass—you have to re-position your hands on the keyboard. This is where *key-binding* comes into play. Using the command `bind`, you can map a Readline Function to any key combination you like. Of course, you should be careful not to overwrite pre-existing key bindings that you want to use. I like to map the following keys to these Readline Functions:
Cntrl-forward-arrow - forward-word
Cntrl-backward-arrow - backward-word
up-arrow - history-search-backward
down-arrow - history-search-forward
In my *.bash_profile* (or, more accurately, in my global bash settings file) I use:

```
# make cursor jump over words
bind '"\e[5C": forward-word'    # control+arrow_right
bind '"\e[5D": backward-word'   # control+arrow_left

# make history searchable by entering the beginning of command
# and using up and down keys
bind '"\e[A": history-search-backward'  # arrow_up
bind '"\e[B": history-search-forward'   # arrow_down
```

(Although these may not work universally.) How does this cryptic symbology translate into these particular keybindings? Again, refer to An Introduction to the Command-Line (on Unix-like systems).

alias, unalias TOP VIEW AS PAGE

As mentioned in An Introduction to the Command-Line (on Unix-like systems) - Aliases, Functions, aliasing is a way of mapping one word to another. It can reduce your typing burden by making a shorter expression stand for a longer one:

```
alias c="cat"
```

This means every time you type a *c* the terminal is going to read it as *cat*, so instead of writing:

```
$ cat file.txt
```

you just write:

```
$ c file.txt
```

Another use of `alias` is to weld particular flags onto a command, so every time the command is called, the flags go with it automatically, as in:

```
alias cp="cp -R"
```

or

```
alias mkdir="mkdir -p"
```

Recall the former allows you to copy directories as well as files, and the later allows you to make nested directories. Perhaps you always want to use these options, but this is a tradeoff between convenience and freedom. In general, I prefer to use new words for aliases and not overwrite preexisting bash commands. Here are some aliases I use in my setup file:

```
# coreutils
alias c="cat"
alias e="clear"
alias s="less -S"
alias l="ls -hl"
alias lt="ls -hlt"
alias ll="ls -al"
alias rr="rm -r"
alias r="readlink -m"
alias ct="column -t"
alias ch="chmod -R 755"
alias chh="chmod -R 644"
alias grep="grep --color"
alias yy="rsync -azv --progress"
# remove empty lines with white space
alias noempty="perl -ne '{print if not m/^(\s*)$/}'"
```

```
# awk
alias awkf="awk -F'\t'"
alias length="awk '{print length}'"
# notesappend, my favorite alias
alias n="history | tail -2 | head -1 | tr -s ' ' | cut -d' ' -f3- | awk '{prin
```

```
# HTML-related
alias htmlsed="sed 's|\&|\&amp\;|g; s|>|\&gt\;|g; s|<|\&lt\;|g;'"
```

```
# git
alias ga="git add"
alias gc="git commit -m"
alias gac="git commit -a -m"
alias gp="git pull"
alias gpush="git push"
alias gs="git status"
alias gl="git log"
alias gg="git log --oneline --decorate --graph --all"
# alias gg="git log --pretty=format:'%h %s %an' --graph"
alias gb="git branch"
alias gch="git checkout"
alias gls="git ls-files"
alias gd="git diff"
```

To display all of your current aliases:

```
$ alias
```

To get rid of an alias, use `unalias`, as in:

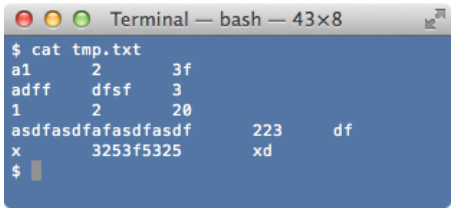
```
$ unalias myalias
```

column TOP VIEW_AS_PAGE

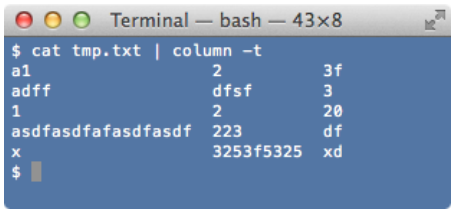
Suppose you have a file with fields of variable length. Viewing it in the terminal can be messy because, if a field in one row is longer than one in another, it will upset the spacing of the columns. You can remedy this as follows:

```
cat myfile.txt | column -t
```

This puts your file into a nice table, which is what the *-t* flag stands for. Here's an illustration:



The file *tmp.txt* is tab-delimited but, because the length of the fields is so different, it looks ugly.
With `column -t`:



If your file has many columns, the `column` command works particularly well in combination with:

```
less -S
```

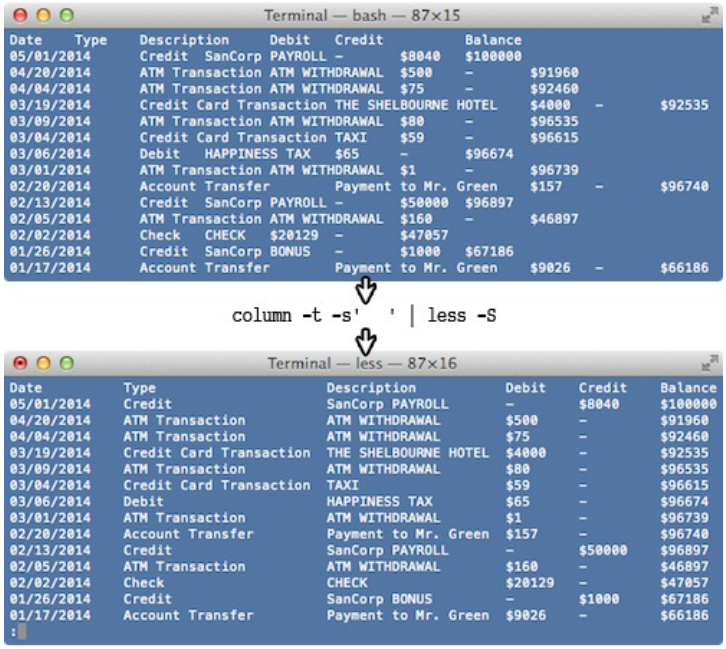
which allows horizontal scrolling and prevents lines from wrapping onto the next row:

```
cat myfile.txt | column -t | less -S
```

`column` delimits on whitespace rather than tab by default, so if your fields themselves have spaces in them, amend it to:

```
cat myfile.txt | column -s'	' -t | less -S
```

(where you make a tab in the terminal by typing *Cntrl-v tab*). This makes the terminal feel almost like an Excel spreadsheet. Observe how it changes the viewing experience for this file of fake financial data:



find TOP VIEW_AS_PAGE

`find` is for finding files or folders in your directory tree. You can also use it to list the paths of all subdirectories and files of a directory. To find files, you can use the simple syntax:

```
find /some/directory -name "myfile"
```

For instance, to find all files with the extension *.html* in the current directory or any of its sub-directories:

```
$ find ./ -name "*.html"
```

The flag *-iname* makes the command case insensitive:

```
$ find /my/dir -iname "*alex*"
```

This would find any file or directory containing *ALEX*, *Alex*, *alex*, and so on in */my/dir* and its children.

To see the path of every child file and directory from the cwd on down, simply type:

```
$ find
```

This could come into play, for example, when you're trying to clean up your folders. In addition to how big a folder is (remember `du -sh`), you might also want to know if the folder has a gazillion files in it. Try:

```
$ find mydirectory | wc -l
```

Say you want to refine this and ask how many files are in each of *mydirectory*'s child directories. Then use:

```
$ for i in mydirectory/*; do echo "***"$i; find $i | wc -l; done
```

All this is to say, `find` is one of the best tools to print out file paths, irrespective of whether you're looking for a file.

`find` works particularly well in combination with `xargs`. For example, delete any files called *tmp.txt* in *mydirectory*:

```
$ find mydirectory -name "tmp.txt" | xargs rm
```

You could also, of course, do this in a plain loop:

```
$ for i in $( find mydirectory -name "tmp.txt" ); do echo $i; rm $i; done
```

Zip any file with the extension *.txt* in *mydirectory*:

```
$ find mydirectory -name "*.txt" | xargs gzip
```

Ditto with a loop:

```
$ for i in $( find mydirectory -name "*.txt" ); do echo $i; gzip $i; done
```

touch TOP VIEW_AS_PAGE

`touch` makes an empty file. E.g., to make an empty file called *test*:

```
$ touch test
```

Sometimes you find yourself running this command to see if you have write permission in a particular directory.

diff, comm TOP VIEW_AS_PAGE

`diff` prints out the differences between two files. It's the best way to find discrepancies between files you think are the same. If:

```
$ cat tmp1
a
a
b
c
```

```
$ cat tmp2
a
x
b
c
```

then `diff` catches that line 2 is different:

```
$ diff tmp1 tmp2
2c2
< a
---
> x
```

`comm`, for common, is similar to `diff` but I rarely have occasion to use it.

Note: If you're familiar with git, you know that the diff operation plays a central role in its version control system. Git has its own flavor of diff: `git diff`. For example, to see the difference between commit *c295z17* and commit *e5d6565*:

```
$ git diff c295z17 e5d6565
```

join TOP VIEW_AS_PAGE

`join` joins two sorted files on a common key (the first column by default). If:

```
$ cat tmp1.txt
1       a
2       b
3       c
```

```
$ cat tmp2.txt
2       aa
3       bb
```

Then:

```
$ join tmp1.txt tmp2.txt
2 b aa
3 c bb
```

My lab uses a short Perl script called *tableconcatlines* (written by my co-worker, Vladimir) to do a join without requiring the files to be pre-sorted:

```
#!/usr/bin/env perl

# About:
# join two text files using the first column of the first file as the "key"

# Useage:
# tableconcatlines example/fileA.txt example/fileB.txt

%h = map {/(\S*)\s(.*)/; $1 => $2} split(/\n/, `cat $ARGV[1]`);

open $ifile, '<', $ARGV[0];

while (<$ifile>)
{
        /^(\S*)/;
        chop;
        print $_ . "\t" . $h{$1} . "\n";
}
```

md5, md5sum TOP VIEW_AS_PAGE

Imagine the following scenario. You've just downloaded a large file from the internet. How do you know no data was lost during the transfer and you've made an exact copy of the one that was online?

To solve this problem, let's review of the concept of hashing. If you're familiar with a dict in Python or a hash in Perl, you know that a hash, as a data structure, is simply a way to map a set of unique keys to a set of values. In ordinary life, an English dictionary is a good representation of this data structure. If you know your key is *"cat"* you can find your value is *"a small domesticated carnivorous mammal with soft fur, a short snout, and retractile claws"*, as Google defines it. In the English dictionary, the authors assigned values to keys, but suppose we only have keys and we want to assign values to them. A *hash function* describes a method for how to boil down keys into values. Without getting deep into the theory of hashing, it's remarkable that you can hash, say, text files of arbitrary length into a determined range of numbers. For example, a very stupid hash would be to assign every letter to a number:

```
A -> 1
B -> 2
C -> 3
.
.
.
```

and then to go through the file and sum up all the numbers; and finally to take, say, modulo 1000. With this, we could assign the novels *Moby Dick*, *Great Expectations*, and *Middlemarch* all to numbers between 1 and 1000! This isn't a good hash function because two novels might well get the same number but nevermind—enough of a digression already.

md5 is a hash function that hashes a whole file into a long string. The commands `md5` and `md5sum` do about the same thing. For example, to compute the md5 hash of a file *tmp.txt*:

```
$ md5 tmp.txt
84fac4682b93268061e4adb49cee9788  tmp.txt
```

```
$ md5sum tmp.txt
84fac4682b93268061e4adb49cee9788  tmp.txt
```

This is a great way to check that you've made a faithful copy of a file. If you're downloading an important file, ask the file's owner to provide the md5 sum. After you've downloaded the file, compute the md5 on your end and check that it's the same as the provided one.

`md5` is one of many hashing functions. Another one, for example, is sha1—the unix utility is `sha1sum`—which will be familiar to users of git:

```
$ sha1sum tmp.txt
fbaaa780c23da55182f448e38b1a0677292dde01  tmp.txt
```

tr TOP VIEW_AS_PAGE

`tr` stands for *translate* and it's a utility for replacing characters in text. For example, to replace a period with a newline:

```
$ echo "joe.joe" | tr "." "\n"
joe
joe
```

Change lowercase letters to uppercase letters:

```
$ echo joe | tr "[:lower:]" "[:upper:]"
JOE
```

Find the numberings of columns in a header (produced by the bioinformatics program BLAST):

```
$ cat blast_header
qid     sid     pid     alignmentlength mismatches      numbergap       query_
```

```
$ cat blast_header | tr "\t" "\n" | nl -b a
     1  qid
     2  sid
     3  pid
     4  alignmentlength
     5  mismatches
     6  numbergap
     7  query_start
     8  query_end
     9  subject_start
    10  subject_end
    11  evalue
    12  bitscore
```

`tr`, with the *-d* flag, is also useful for deleting characters. If we have a file *tmp.txt*:

```
$ cat tmp.txt
a a a a
a b b b
a v b b
1 b 2 3
```

then:

```
$ cat tmp.txt | tr -d "b"
a a a a
a
a v
1  2 3
```

This is one of the easiest ways to delete newlines from a file:

```
$ cat tmp.txt | tr -d "\n"
a a a aa b b ba v b b1 b 2 3
```

Tip: To destroy carriage return characters ("\r"), often seen when you open a Windows file in linux, use:

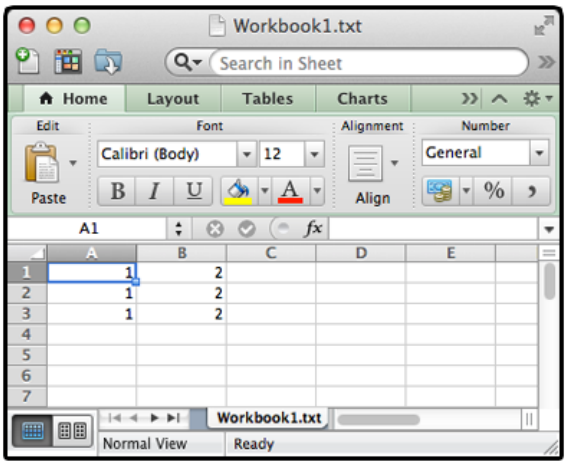
```
$ cat file.txt | tr -d "\r"
```

od TOP VIEW_AS_PAGE

One of the most ninja moves in unix is the `od` command which, with the *-tc* flag, explicitly prints every character in a string or a file. For example:

```
$ echo joe | od -tc
0000000   j   o   e  \n
0000004
```

We see everything: the *j*, the *o*, the *e*, and the newline. This is incredibly useful for debugging, especially because some programs—notably those that bear the Microsoft stamp—will silently insert evil "landmine" characters into your files. The most common issue is that Windows/Microsoft sometimes uses a carriage-return (\r), whereas Mac/unix uses a much more sensible newline (\n). If you're transferring files from a Windows machine to unix or Mac and let your guard down, this can cause unexpected bugs. Consider a Microsoft Excel file:



If we save this spreadsheet as a text file and try to `cat` it, it screws up? Why? Our `od` command reveals the answer:

```
$ cat Workbook1.txt | od -tc
0000000   1  \t   2  \r   1  \t   2  \r   1  \t   2
0000013
```

Horrible carriage-returns! Let's fix it and check that it worked:

```
$ cat ~/Desktop/Workbook1.txt | tr "\r" "\n" | od -tc
0000000   1  \t   2  \n   1  \t   2  \n   1  \t   2
0000013
```

Score!

split TOP VIEW_AS_PAGE

`split` splits up a file. For example, suppose that:

```
$ cat test.txt
1
2
3
4
5
6
7
8
9
10
```

If we want to split this file into sub-files with 3 lines each, we can use:

```
$ split -l 3 -d test.txt test_split_
```

where *-l 3* specifies that we want 3 lines per file; *-d* specifies we want numeric suffixes; and *test_split_* is the prefix of each of our sub-files. Let's examine the result of this command:

```
$ head test_split_*
==> test_split_00 <==
1
2
3

==> test_split_01 <==
4
5
6
```

```
==> test_split_02 <==
7
8
9

==> test_split_03 <==
10
```

Note that the last file doesn't have 3 lines because 10 is not divisible by 3—its line count equals the remainder.

Splitting files is an important technique for parallelizing computations.

nano, emacs, vim TOP VIEW_AS_PAGE

`nano` is a basic a text editor that should come pre-packaged with your linux distribution. There are two fundamental commands you need to know to use nano:
Cntrl-O - Save
Cntrl-X - Quit
nano is not a great text editor and you shouldn't use it for anything fancy. But for simple tasks—say, pasting swaths of text into a file—it's a good choice.

For more serious text editing, use Vim or Emacs. These programs have too many features to discuss in this post but, as a Vim partisan, I have an introduction to Vim here.

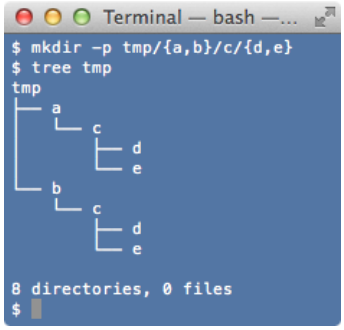
tree TOP VIEW_AS_PAGE

Note: `tree` is not a default shell program. You may have to download and install it.

`tree` prints out a tree, in ASCII characters, of your directory structure. For example:

```
$ mkdir -p tmp/{a,b}/c/{d,e}
```

Let's run `tree` on this directory and illustrate its output:



If you make the mistake of running this command in your home directory, you'll get a monstrously large output. For that reason, you can restrict the depth with the *-L* flag:

```
$ tree -L 2 tmp
```

This prints the tree only two directory levels down.

`tree` is like vanilla `find` with graphics.

screen TOP VIEW_AS_PAGE

Imagine the following scenario: you're working in the terminal and you accidentally close the window or quit or it crashes. You've lost your session. If you were logged in somewhere, you'll have to log in again. If you had commands you wanted to refer back to, they're lost forever. This is where `screen` or `tmux` comes in. Regardless of whether or not you quit the terminal, your sessions are persistent processes that you can always access until the computer is shut off. You can get your exact terminal window back again: you're still logged in wherever you were and, moreover, you can see everything you typed in this window by simply

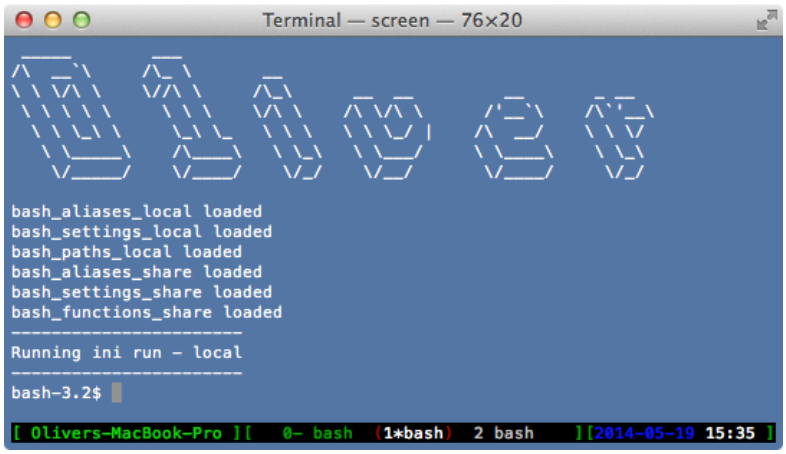
scrolling up. For this reason, it's a good idea to make using `screen` or `tmux` a habit every time you're on the terminal, if only as insurance. `screen` also allows you to access multiple terminal instances within the same terminal window and more.

A word of advice: skip the rest of this section. Learn `tmux`, the superior program (discussed next), instead of wasting brain space on `screen`.

Start `screen`:

```
$ screen
```

Let's see an example of what this looks like:



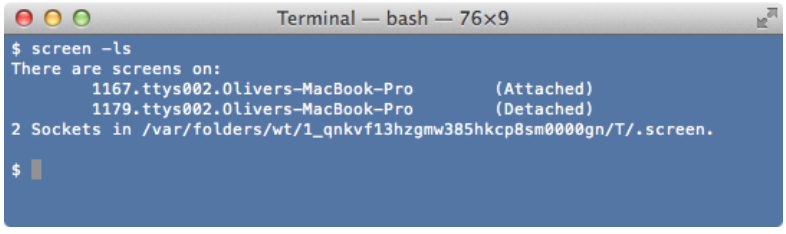
This looks like an ordinary terminal window except that you see a bar at the bottom with some names:
0 bash
1 bash
2 bash
These are all the terminal instances we have access to in this one screen session (and in this one window).

So, let's get this straight: we can have many different screen sessions and, within each session, we can have multiple instances which are—muddying the terminology—also called *windows* (but *not* the kind of window you can create on the Mac by typing ⌘*N* which I was referring to above).

List current screen sessions:

```
$ screen -ls
```

Another figure:



To reattach—i.e., enter into—screen session of id *1167*:

```
$ screen -r 1167
```

To kill a screen session of id *1167*:

```
$ screen -X -S 1167 quit
```

Once you're in a screen session, what are the commands? By default, each screen command starts with *Cntrl-a*, but I find this impractical because this combination's default key binding in bash is to jump your cursor to the beginning of the line, which I use all the time. So I like to use *Cntrl-f* as my Leader Key. You can use whatever you like—we'll see how to set it below with *.screenrc*. Let's learn some of the basic screen commands. I will copy some of these directly from here:

<Leader> c	Create new window (shell)
<Leader> k	Kill the current window
<Leader> n	Go to the next window
<Leader> p	Go to the previous window
<Leader> <Leader>	Toggle between the current and previous window
<Leader> d	Detach the screen session
<Leader> A	Set window's title
<Leader> w	List all windows
<Leader> 0-9	Go to a window numbered 0-9
<Leader> [	Start copy mode
<Leader> Esc	Start copy mode
<Leader>]	Paste copied text
<Leader> ?	Help (display a list of commands)

A note: to scroll inside screen (you'll find you can't use the window scrollbar anymore), enter into copy mode and then navigate up and down with *Cntrl-u* and *Cntrl-d* or *j* and *k*, as in Vim.

You can configure `screen` with the file *.screenrc*, placed in your home directory. Here's mine, which my co-worker, Albert, "lent" me:

```
# change Leader key to Cntrl-f rather than Cntrl-a
escape ^Ff

defscrollback 5000
shelltitle '$ |bash'
autodetach on

# disable the startup splash message that screen displays on startup
startup_message off

# create a status line at the bottom of the screen.  this will show the titles and locat
# all screen windows you have open at any given time
hardstatus alwayslastline '%{= kG}[ %{G}%H %{g}][%= %{=kw}%?%-Lw%?%{r}(%{W}%n*%f%t%?(%u)
```

This demonstrates how to use *Cntrl-f* as the Leader key, rather than *Cntrl-a*.

tmux TOP VIEW AS PAGE

Note: `tmux` is not a default shell program. You may have to download and install it.

As discussed in An Introduction to the Command-Line (on Unix-like systems) - tmux, `tmux` is a more modern version of `screen`, which all the cool kids are using. In `tmux` there are:

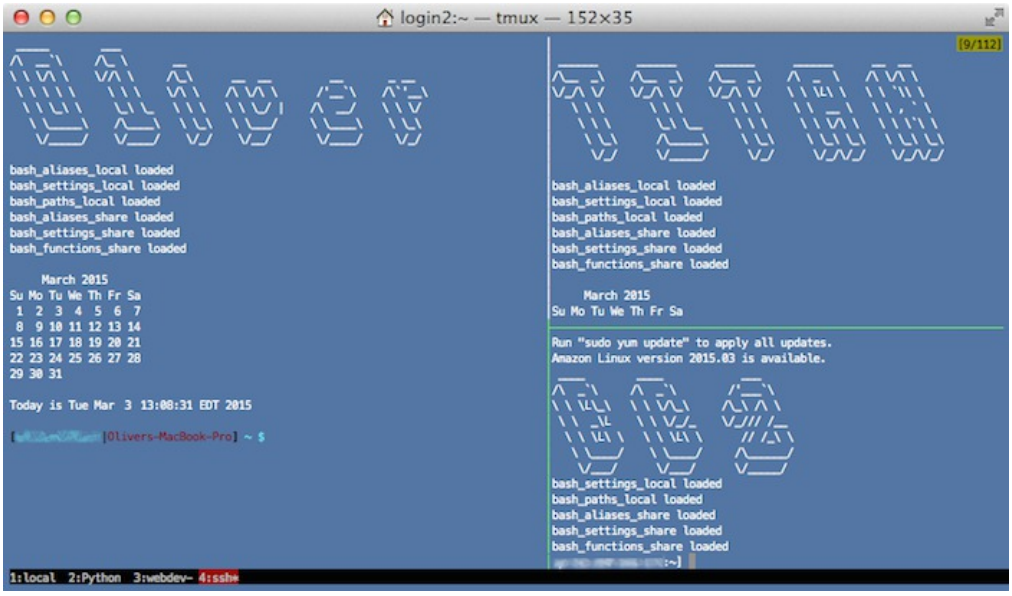
sessions

windows

panes

Sessions are groupings of `tmux` *windows*. For most purposes, you only need one session. Within each session, we can have multiple "windows" which you see as tabs on the bottom of your screen (i.e., they're virtual windows, not the kind you'd create on a Mac by typing ⌘*N*). You can further subdivide windows into

panes, although this can get hairy fast. Here's an example of a `tmux` session containing four windows (the tabs on the bottom), where the active window contains 3 panes:



I'm showing off by logging into three different computers—home, work, and Amazon.

Start a new session:

```
$ tmux
```

Detach from your session: *Leader d*, where the leader sequence is *Cntrl-b* by default.

List your sessions:

```
$ tmux ls
0: 4 windows (created Fri Sep 12 16:52:30 2014) [177x37]
```

Attach to a session:

```
$ tmux attach                   # attach to the first session
$ tmux attach -t 0              # attach to a session by id
```

Kill a session:

```
$ tmux kill-session -t 0        # kill a session by id
```

Here's a more systematic list of commands:

<Leader> c	Create a new window
<Leader> x	Kill the current window
<Leader> n	Go to the next window
<Leader> p	Go to the previous window
<Leader> l	Go to the last-seen window
<Leader> 0-9	Go to a window numbered 0-9
<Leader> w	List windows
<Leader> s	Toggle between sessions
<Leader> d	Detach the session
<Leader> ,	Set a window's title

<Leader> [	Start copy mode (which will obey vim conventions per order of the config file)
<Leader>]	Paste what you grabbed in copy mode
<Leader> %	Spawn new horizontal pane (within the window)
<Leader> "	Spawn new vertical pane (within the window)
<Leader> o	Go to the next pane (within the window)
<Leader> o *(pressed simultaneously*)*	Swap position of panes (within the window)
<Leader> q	Number the panes (within the window) - whereupon you can jump to a specific pane by pressing its numerical index
<Leader> z	Toggle the expansion of one of the panes (within the window)
<Leader> t	Display the time
<Leader> m	Turn mouse mode on (allows you to resize panes with the mouse)
<Leader> M	Turn mouse mode off
<Leader> :	Enter command mode

* With `tmux`, in contrast to `screen`, you give a slightly longer pause after pressing the Leader sequence. In this particular case, however, they are pressed simultaneously.

As before, I like to use *Cntrl-f* as my Leader sequence. Here's a sample *.tmux.conf* file, which accomplishes this, among other useful things:

```
### Default Behavior
# Use screen key binding
set-option -g prefix C-f

# 1-based numbering
set -g base-index 1

# Fast command sequence - cause some error
# set -s escape-time 0

# Set scrollback to 10000 lines
set -g history-limit 10000

# Agressive window
```

To reload *.tmux.conf* after making changes: *<Leader> :source-file ~/.tmux.conf*

To copy and paste text, assuming vi bindings:
first enter copy mode with *<Leader> [*
copy text with *space* to enter visual mode, then select text, then use *y* to yank, then hit *Enter*
paste text with *<Leader>]*
Occasionally, you may find yourself using `tmux` within `tmux`. Suppose you're running a `tmux` session on your current computer and in one of your windows you `ssh` into another computer, which is also running `tmux`. In this window, you run:

```
$ tmux attach
```

and now you're running `tmux` within `tmux`. To execute `tmux` commands on the inner session, use *Cntrl-b* after the *<Leader>* sequence. For example, to list your windows on the inner session, you'd type *<Leader> Cntrl-b w*.

`tmux` is amazing—I've gone from newbie to cheerleader in about 30 seconds!

Hat tip: Albert

make TOP VIEW_AS_PAGE

`make` is a program which executes a special type of script called a *Makefile*. As this tutorial notes:

> As a build automation tool, Make generates files (called targets), each of which can depend upon the existence of other files (called dependencies). Targets, dependencies, and instructions for how to build them (called recipes) are defined in a special file called a Makefile.

Suppose you have a file called *input1* and you want to produce a file called *output3*. You write a makefile, which species how to do this with a bunch of rules. In pseudocode:
Rule1: Use input1 to produce output1
Rule2: Use output1 to produce output2
Rule3: Finally, use output2 to produce output3
You run `make` and produce your *output3*. At this point you're asking, why didn't I just use an ordinary script? Suppose now you delete *output3* and want to produce it anew. The beauty of `make` is that it works backwards from your final rule, figures out which dependencies it needs to create your final file, and only runs the necessary rules. In our example, if *output2* were still present, it would only run *Rule3*. If `make` detected *output2* were missing, it would look to run *Rule2*, etc.

I'll let the experts speak here, since I don't have much experience with this utility:
Why Use Make by Mike Bostock
Making Data, the DataMade Way by hancush
How ProPublica Illinois Uses GNU Make to Load 1.4GB of Data Every Day
Note: The make concept—a script that starts backwards and only runs what it needs to—is so useful it's been adapted into a Python version called Snakemake primarily used for bioinformatics.

yes TOP VIEW_AS_PAGE

`yes` prints out the character *y* in an infinite loop (so be careful - you'll have to stop it with *Cntrl-c*). If you give `yes` an argument, it will print out the argument rather than *y*. For instance, to print the word *oliver* 30 times, the command would be:

```
$ yes oliver | head -30
```

nl TOP VIEW_AS_PAGE

`nl` numbers each row in a file. E.g.:

```
$ cat tmp.txt
aaa
bbb
ccc
ddd
eee
```

```
$ nl -b a tmp.txt
     1  aaa
     2  bbb
     3  ccc
     4  ddd
     5  eee
```

You can accomplish a similar thing with `awk`:

```
$ cat tmp.txt | awk '{print NR"\t"$0}'
1       aaa
2       bbb
3       ccc
4       ddd
5       eee
```

whoami TOP VIEW_AS_PAGE

To see your unix username:

```
$ whoami
```

groups TOP VIEW_AS_PAGE

To see what groups you belong to:

```
$ groups
```

who, w TOP VIEW_AS_PAGE

`who` and `w` are very similar. Both show a list of users currently logged on to the system, but `w` gives slightly more information.

See users currently logged in:

```
$ who
```

See users currently logged in as well as what command they're running:

```
$ w
```

hostname TOP VIEW_AS_PAGE

`hostname` prints the system's host name, which is like the name of the computer. If you've `ssh`-ed into multiple computers in a bunch of different terminal sessions, this command will remind you where you are:

```
$ hostname
```

finger TOP VIEW_AS_PAGE

`finger` is a neat command that prints out information about a user on the system. For example, to `finger` yourself:

```
$ finger $( whoami )
Login: my_username                         Name: My Name
Directory: /path/to/home Shell: /bin/bash
Never logged in.
No mail.
Plan:
Hello, fools and lovers!
```

What's the plan thing? When you `finger` a user it will output the contents of the file

`~/.plan`

(Was this the genesis of Amherst College's PlanWorld? I think so! :-)

read TOP VIEW_AS_PAGE

`read` is a shell builtin that I like to use in loops. You can read up a file with a while loop using `read`—e.g., spit out *file.txt* exactly as is:

```
$ while read x; do echo "$x"; done < file.txt
```

Another example:

```
$ echo -e '1\t2\n3\t4\n5\t6'
1       2
3       4
5       6
$ echo -e '1\t2\n3\t4\n5\t6' | while read x y; do echo $x; done
1
```

```
3
5
```

tee TOP VIEW_AS_PAGE

As discussed in An Introduction to the Command-Line (on Unix-like systems) - Piping in Unix, `tee` is a command sometimes seen in unix pipelines. Suppose we have a file *test.txt* such that:

```
$ cat test.txt
1       c
3       c
2       t
1       c
```

Then:

```
$ cat test.txt | sort -u | tee tmp.txt | wc -l
3
```

```
$ cat tmp.txt
1       c
2       t
3       c
```

`tee`, in rough analogy with a plumber's tee fitting, allows us to save a file in the middle of the pipeline and keep going. In this case, the output of `sort` is both saved as *tmp.txt* and passed through the pipe to `wc -l`, which counts the lines of the file.

Another similar example:

```
$ echo joe | tee test.txt
joe
```

```
$ cat test.txt
joe
```

The same idea: *joe* is echoed to std:out as well as saved in the file *test.txt*.

shopt TOP VIEW_AS_PAGE

`shopt`, for *shell options*, controls various togglable shell options, as its name suggests. You can see the options it controls by simply entering the command. For example, on my system:

```
$ shopt
cdable_vars     off
cdspell         off
checkhash       off
checkwinsize    off
cmdhist         on
compat31        off
dotglob         off
.
.
.
```

There's a discussion about `shopt`'s *histappend* option in An Introduction to the Command-Line (on Unix-like systems) - history.

true, false TOP VIEW_AS_PAGE

`true` and `false` are useful to make multi-line comments in a bash script:

```
# multi-line comment
if false; then
echo hello
```

```
echo hello
echo hello
fi
```

To comment these `echo` statements back in:

```
if true; then
echo hello
echo hello
echo hello
fi
```

The formal man page definitions of these commands are amusing: `true`: *"do nothing, successfully (exit with a status code indicating success)"*; `false`: *"do nothing, unsuccessfully (exit with a status code indicating failure)"*.

shift TOP VIEW_AS_PAGE

As in Perl and many other languages, `shift` pops elements off the array of input arguments in a script. Suppose *tmpscript* is:

```
#!/bin/bash
echo $1
shift
echo $1
```

Then:

```
$ ./tmpscript x y
x
y
```

Refer to An Introduction to the Command-Line (on Unix-like systems) - Arguments to a Script to see how to use `shift` to make an input flag parsing section for your script.

g++ TOP VIEW_AS_PAGE

Compile a C++ program, *myprogram.cpp*:

```
$ g++ myprogram.cpp -o myprogram
```

This produces the executable *myprogram*. Stackoverflow has a nice post about the `g++` optimization flags, which includes the bit:

> The rule of thumb:
> When you need to debug, use -O0 (and -g to generate debugging symbols.)
> When you are preparing to ship it, use -O2.
> When you use gentoo, use -O3...!
> When you need to put it on an embedded system, use -Os (optimize for size, not for efficiency.)

I usually use *-O2*:

```
$ g++ -O2 myprogram.cpp -o myprogram
```

Read about GCC, the GNU Compiler Collection, here.

xargs TOP VIEW_AS_PAGE

`xargs` is a nice shortcut to avoid using for loops. For example, let's say we have a bunch of *.txt* files in a folder and they're symbolic links we want to read. The following two commands are equivalent:

```
$ for i in *.txt; do readlink -m $i; done
$ ls *.txt | xargs -i readlink -m {}
```

In `xargs` world, the *{}* represents "the bucket"—i.e., what was passed through the pipe.

Here are some more examples. Make symbolic links *en masse*:

```
$ ls /some/path/*.txt | xargs -i ln -s {}
```

`grep` a bunch of stuff *en masse*:

```
$ # search for anything in myfile.txt in anotherfile.txt
$ cat myfile.txt | xargs -i grep {} anotherfile.txt
```

`xargs` works particularly well with `find`. For example, zip all the files with extension .fastq (bioinformatics) found in a directory or any of its sub-directories:

```
$ find my_directory -name "*.fastq" | xargs -i gzip {}
```

Delete all pesky .DS_store files in the cwd or any of its sub-directories:

```
$ find ./ -name ".DS_Store" | xargs -i rm {}
```

crontab TOP VIEW_AS_PAGE

`crontab` is a tool to automatically run a command or script at periodic intervals. The first hit on Google has a nice how-to about crontab:

http://www.thesitewizard.com/general/set-cron-job.shtml

To quote directly from this source, you need to make a file of the form:

Field	Range of Values
minute	0-59
hour	0-23
day	1-31
month	1-12
day-of-week	0-7 (where both 0 and 7 mean Sun, 1 = Mon, 2 = Tue, etc)
command-line-to-execute	the command to run along with the parameters to that command if any

Once you make a file of this form, you can run it. For example, if:

```
$ cat mycron.txt
45 10 * * * rm -r /full/path/trash/* > /full/path/out.o 2> /full/path/out.e
```

Then you run this as:

```
$ crontab mycron.txt
```

And it will empty the trash folder every day at 10:45 a.m.

Display your cron jobs:

```
$ crontab -l
```

One peculiar thing about cron is that, if you're on a shared computer system, your system administrator might be the one running cron in his name and, when you invoke cron, you won't start a process running under your name. For this reason, you have to use absolute paths in the commands you put in your cron job: cron doesn't know about the local folder from which you run the command.

type TOP VIEW_AS_PAGE

When you enter some text at the terminal prompt, that text could be a command (a binary or a script), an alias, a bash function, a bash keyword, a bash builtin, and so on. `type` will tell you, among these, what type of entity you're dealing with. For example, on my system:

```
$ type ssh
ssh is /usr/bin/ssh
```

This is a straight-up command, not an alias or function. But, if I've aliased *quickssh* to some specific ssh address, then `type` will reveal this:

```
$ type quickssh
quickssh is aliased to `ssh -Y user@myserver.edu'
```

Also, observe the following:

```
$ type if
if is a shell keyword
```

```
$ type set
set is a shell builtin
```

`type` tells us that `if` is a reserved word in bash, while `set` is a built-in command. You get the idea!

info TOP VIEW_AS_PAGE

The `info` command has a wealth of information about, well, nearly everything:

```
$ info
```

To get information about, say, the `grep` command:

```
$ info grep
```

apropos TOP VIEW_AS_PAGE

The `apropos` command will display commands related to the string you give it as an argument. For instance:

```
$ apropos grep
bzegrep [bzgrep]     (1)  - search possibly bzip2 compressed files for a regular expression
bzfgrep [bzgrep]     (1)  - search possibly bzip2 compressed files for a regular expression
bzgrep               (1)  - search possibly bzip2 compressed files for a regular expression
egrep [grep]         (1)  - print lines matching a pattern
fgrep [grep]         (1)  - print lines matching a pattern
git-grep             (1)  - Print lines matching a pattern
grep                 (1)  - print lines matching a pattern
grep                 (1p)  - search a file for a pattern
pgrep                (1)  - look up or signal processes based on name and other attributes
pkill [pgrep]        (1)  - look up or signal processes based on name and other attributes
ptargrep             (1)  - Apply pattern matching to the contents of files in a tar archive
xzgrep               (1)  - search compressed files for a regular expression
xzgrep [lzegrep]     (1)  - search compressed files for a regular expression
xzgrep [lzfgrep]     (1)  - search compressed files for a regular expression
xzgrep [lzgrep]      (1)  - search compressed files for a regular expression
xzgrep [xzegrep]     (1)  - search compressed files for a regular expression
xzgrep [xzfgrep]     (1)  - search compressed files for a regular expression
zgrep                (1)  - search possibly compressed files for a regular expression
zipgrep              (1)  - search files in a ZIP archive for lines matching a pattern
```

fold TOP VIEW_AS_PAGE

The `fold` command allows you to restrict the number of characters per line written to std:out. For example:

```
$ echo -e "asdfasdfasdf\nasdfasdfasdf"
asdfasdfasdf
asdfasdfasdf
```

Using a 5 for the *width* flag produces:

```
$ echo -e "asdfasdfasdf\nasdfasdfasdf" | fold -w 5
asdfa
sdfas
df
asdfa
sdfas
df
```

Transpose a string (i.e., convert from row to column):

```
$ echo Joe | fold -w 1
J
o
e
```

Here's an example from Wiki - Bioinformatics. How would you find the sequence composition of the following fasta file?

```
$ cat myfasta.fa
>entry1
AACCCGGCTGCGTACGTACCACAGAGAGGGGTGTA
>entry2
TTATGCGATAAACCCGGGTGTAATTTTATTTTTTT
```

Here's the answer:

```
$ cat myfasta.fa | grep -v ">" | fold -w 1 | sort | uniq -c
     17 A
     13 C
     18 G
     22 T
```

rev TOP VIEW_AS_PAGE

`rev` reverses a string:

```
$ echo hello | rev
olleh
```

As noted in the bioinformatics wiki, we can use this trick to find the *reverse complement* of a string representing DNA nucleotides:

```
$ echo CACTTGCGGG | rev | tr ATGC TACG
CCCGCAAGTG
```

mount TOP VIEW_AS_PAGE

`mount`, as the docs say, mounts a filesystem. Imagine two directory trees, one on your computer and one on an external hard drive. How do you graft these trees together? This is where `mount` comes in. On your Macintosh, external hard drives get mounted by default in the

`/Volumes`

directory of your computer. Put differently, the directory tree of your computer and the directory tree of your HD are fused there. I've only ever used the actual `mount` command in the context of mounting an EBS volume on an EC2 node on Amazon Web Services.

mktemp TOP VIEW_AS_PAGE

`mktemp` will make a temporary directory with a unique name in your designated temporary directory. Recall that, usually, this is the directory:

```
/tmp
```

However, you can set it to be anything you like using the variable `TMPDIR`. Let's see `mktemp` in action:

```
$ echo $TMPDIR
/path/tempdir
$ mktemp
/path/tempdir/tmp.LaOYagQwq3
$ mktemp
/path/tempdir/tmp.h5FapGH4LS
```

`mktemp` spits out the name of the directory it creates.

watch TOP VIEW_AS_PAGE

If you prefix a command with `watch`, it will repeat every 2.0 seconds and you can monitor it. For instance, if a file in */some/directory* is growing, and you want to monitor the directory's size:

```
$ watch ls -hl /some/directory
```

Sometimes you want to monitor how much space you have left on your computer:

```
$ watch df -h .
```

If a program is churning away and slowly writing to a log file, you can watch the tail:

```
$ watch tail log.txt
```

Another example is using `watch` in combination with the Oracle Grid Engine's `qstat` to monitor your job status:

```
$ watch qstat
```

perl, python TOP VIEW_AS_PAGE

Perl and Python aren't really unix commands, but whole massive programming languages in themselves. Still, there are some neat things you can do with them on the command line. See:

An Introduction to the Command-Line (on Unix-like systems) - Command Line Perl and Regex

An Introduction to the Command-Line (on Unix-like systems) - Using the Python Shell to do Math

ping TOP VIEW_AS_PAGE

As the docs say, `ping` *"uses the ICMP protocol's mandatory ECHO_REQUEST datagram to elicit an ICMP ECHO_RESPONSE from a host or gateway."* Practically speaking, it allows you to send a signal to some IP address to see if it's responding. Think of it as an exchange between two people:

"Hey, are you there?"
"Yes, I'm here."

The syntax is:

```
$ ping some_IP_address
```

For instance, the internet tells me that *74.125.224.18* belongs to Google, so:

```
$ ping 74.125.224.18
PING 74.125.224.18 (74.125.224.18) 56(84) bytes of data.
64 bytes from 74.125.224.18: icmp_seq=1 ttl=43 time=76.5 ms
64 bytes from 74.125.224.18: icmp_seq=2 ttl=43 time=76.7 ms
64 bytes from 74.125.224.18: icmp_seq=3 ttl=43 time=76.7 ms
64 bytes from 74.125.224.18: icmp_seq=4 ttl=43 time=76.6 ms
```

```
64 bytes from 74.125.224.18: icmp_seq=5 ttl=43 time=76.7 ms
^C
--- 74.125.224.18 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4414ms
rtt min/avg/max/mdev = 76.594/76.700/76.793/0.359 ms
```

(Interrupt `ping` with *Cntrl-c*.) You can see this on the network. However, if you try to `ping` the address *00.000.000.00*:

```
$ ping 00.000.000.00
PING 00.000.000.00 (0.0.0.0): 56 data bytes
ping: sendto: No route to host
ping: sendto: No route to host
Request timeout for icmp_seq 0
ping: sendto: No route to host
Request timeout for icmp_seq 1
ping: sendto: No route to host
Request timeout for icmp_seq 2
ping: sendto: No route to host
Request timeout for icmp_seq 3
^C
--- 00.000.000.00 ping statistics ---
5 packets transmitted, 0 packets received, 100.0% packet loss
```

you'll discover it's dead.

dig TOP VIEW_AS_PAGE

`dig` is the *"DNS lookup utility."* If we use the same test IP address as we did in the example above:

```
$ dig 74.125.224.18

; <<>> DiG 9.8.2rc1-RedHat-9.8.2-0.23.rc1.32.amzn1 <<>> 74.125.224.18
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 59017
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 0

;; QUESTION SECTION:
;74.125.224.18.                 IN      A

;; AUTHORITY SECTION:
.                       10467   IN      SOA     \
a.root-servers.net. nstld.verisign-grs.com. 2014052201 1800 900 604800 86400

;; Query time: 1 msec
;; SERVER: 172.16.0.23#53(172.16.0.23)
;; WHEN: Thu May 22 23:10:06 2014
;; MSG SIZE  rcvd: 106
```

This is the IP's DNS record.

ifconfig TOP VIEW_AS_PAGE

`ifconfig` spits out a bunch of network-related information, often including your IP address and MAC address:

```
$ ifconfig
```

Your IP address should be somewhere around here:

```
$ ifconfig | grep inet
```

If you're looking for your MAC address, it should be labeled as *HWaddr*. I'm the opposite of a network guru and am ignorant about the many other capabilities of this command, but you can read `ifconfig`'s Wikipedia entry.

wget TOP VIEW_AS_PAGE

`wget` is a tool for downloading files from the web. If the movie *test.MOV* is hosted on *example.com*, you can grab it with `wget`:

```
$ wget http://example.com/test.MOV
```

Of course, you can batch this using a loop. Suppose that there are a bunch of files on *example.com* and you put their names in a text file, *list.txt*. Instead of surfing over to the page and having to click each link in your browser, you can do:

```
$ cat list.txt | while read i; do echo $i; wget "http://example.com/"${i}; don
```

or, equivalently:

```
$ while read i; do echo $i; wget "http://example.com/"${i}; done < list.txt
```

As a concrete example, to get and untar the latest version (as of this writing) of the GNU Coreutils, try:

```
$ wget http://ftp.gnu.org/gnu/coreutils/coreutils-8.23.tar.xz
$ tar -xvf coreutils-8.23.tar.xz
```

You can also use `wget` to download a complete offline copy of a webpage. This page in Linux Journal describes how. I will quote their example verbatim:

```
$ wget \
   --recursive \
   --no-clobber \
   --page-requisites \
   --html-extension \
   --convert-links \
   --restrict-file-names=windows \
   --domains website.org \
   --no-parent \
       www.website.org/tutorials/html/
```

This command downloads the Web site www.website.org/tutorials/html/.

The options are:
--recursive: download the entire Web site.
--domains website.org: don't follow links outside website.org.
--no-parent: don't follow links outside the directory tutorials/html/.
--page-requisites: get all the elements that compose the page (images, CSS and so on).
--html-extension: save files with the .html extension.
--convert-links: convert links so that they work locally, off-line.
--restrict-file-names=windows: modify filenames so that they will work in Windows as well.
--no-clobber: don't overwrite any existing files (used in case the download is interrupted and resumed).

elinks, curl TOP VIEW_AS_PAGE

Note: `elinks` is not a default shell program. You may have to download and install it.

`elinks` is a terminal-based web browser. Surfing the web from the terminal is novel, if not terribly convenient.

Go to The New York Times website:

```
$ elinks www.nytimes.com
```

For example, here's what Google looks like in my pink terminal:

More usefully, `elinks` provides a way to scrape webpages into text files:

```
$ elinks www.nytimes.com > dump1.txt
```

If you want the actual html code, you can use `curl`:

```
$ curl www.nytimes.com > dump2.txt
```

Read more about curl on its homepage.

As noted in the Vim wiki, you can also examine the HTML source code of a page right from your terminal with Vim:

```
$ vim http://www.nytimes.com/
```

apt-get, brew, yum TOP VIEW_AS_PAGE

From An Introduction to the Command-Line (on Unix-like systems) - Installing Programs on the Command Line: In the course of this article, we've made passing references to programs like `htop`, `pstree`, `nginx`, and `tmux`. Perhaps you have these on your computer but, if not, how do you get them? You can usually go to a program's webpage, download the source code, and build it on your computer (google the mantra `./configure; make; make install`). However, you'll probably discover that your system has some crucial deficiency. Something's out of date, plus the program you're trying to install requires all these other programs which you don't have. You've entered what I call *dependency hell*, which has a way of sending you to obscure corners of the internet to scour chat rooms for mysterious lore.

When possible, avoid this approach and instead use a *package manager*, which takes care of installing a program—and all of its dependencies—for you. Which package manager you use depends on which operating system you're running. For Macintosh, it's impossible to live without `brew`, whose homepage calls it, *"The missing package manager for OS X."* For Linux, it depends on your distribution's lineage: Ubuntu has `apt-get`; Fedora has `yum`; and so on. All of these package managers can search for packages—i.e., see what's out there—as well as install them.

Let's use the program `gpg2` (The GNU Privacy Guard), a famous data encryption tool which implements the OpenPGP standard, to see what this process looks like. First, let's search:

```
$ brew search gpg2                    # Mac
$ yum search gpg2                     # Fedora
$ apt-cache search gpg2               # Ubuntu
```

Installing it might look like this:

```
$ brew install gpg2                     # Mac
$ sudo yum install gnupg2.x86_64        # Fedora
$ sudo apt-get install gpgv2            # Ubuntu
```

The exact details may vary on your computer but, in any case, now you're ready to wax dangerous and do some Snowden-style hacking! (He reportedly used this tool.)

display, convert, identify TOP VIEW_AS_PAGE

Note: `display`, `convert`, and `identify` are not default shell programs. You have to download and install them from ImageMagick, an awesome suite of command-line tools for manipulating images.

`display` is a neat command to display images right from your terminal via the X Window System:

```
$ display my_pic.png
```

`identify` gets properties about an image file:

```
$ identify my_pic.png
my_pic.png PNG 700x319 700x319+0+0 8-bit DirectClass 84.7kb
```

So, this is a 700 by 319 pixel image in PNG format and so on.

`convert` is a versatile command that can do about 8 gzillion things for you. For example, resize an image:

```
$ convert my_pic.png -resize 550x550\> my_pic2.png
```

Add whitespace around the image file *my_pic.png* until its 600px by 600px and convert it into gif format:

```
$ convert my_pic.png -background white -gravity center -extent 600x600 my_pic.gif
```

Do the same thing, but turn it into jpg format and put it at the top of the new image file (*-gravity north*):

```
$ convert my_pic.png -background white -gravity north -extent 600x600 my_pic.jpg
```

Change an image's width (in this case from 638 to 500 pixels) while preserving its aspect ratio:

```
$ convert my_pic_638_345.png -resize 500x my_pic_500.png
```

As a common use case, the ImageMagick commands are invaluable for optimizing images you're using on your websites. See How to Make a Website - Working with and Optimizing Images for Your Site for details.

gpg TOP VIEW_AS_PAGE

Note: `gpg` is not a default shell program. You have to download and install it.

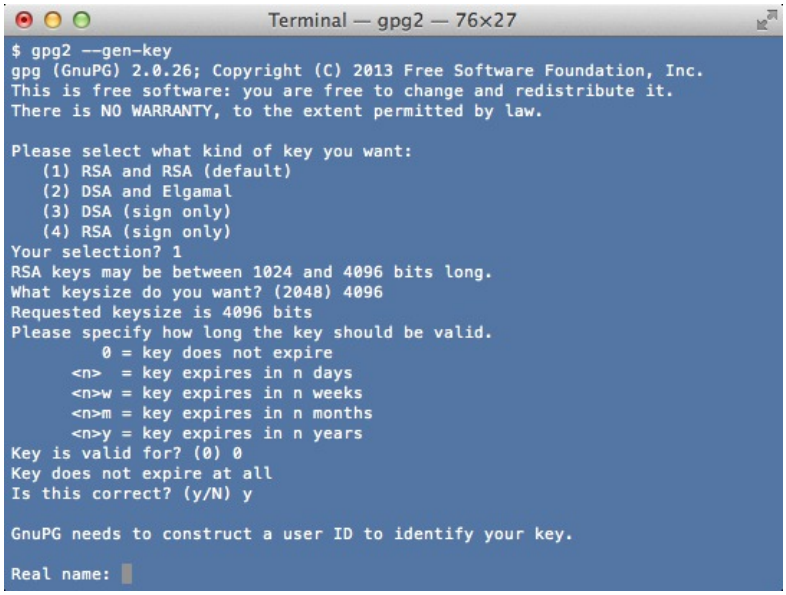
`gpg`, *"a complete and free implementation of the OpenPGP standard,"* according to the official docs, is a program for encrypting files. You can use it to implement public-key cryptography, which Wikipedia describes as follows:

> Public-key cryptography, or asymmetric cryptography, is a cryptographic system that uses pairs of keys: public keys, which may be disseminated widely, and private keys, which are known only to the owner. The generation of such keys depends on cryptographic algorithms based on mathematical problems to produce one-way functions. Effective security only requires keeping the private key private; the public key can be openly distributed without compromising security.
>
> In such a system, any person can encrypt a message using the receiver's public key, but that encrypted message can only be decrypted with the receiver's private key.

Here are the basics about `gpg`. The first thing to do is to generate a public/private *key pair* for yourself:

```
$ gpg --full-generate-key
```

The process will look something like this:



Since you're already going through the trouble, choose a 4096 bit key and a secure passphrase (you're going to need this passphrase in the future to decrypt stuff, so remember it). Provide your real name and email, which will be associated with the key pair.

If you want to generate keys with default options, just use:

```
$ gpg --gen-key
```

To encrypt a message (or file, or tarball) for somebody, you need to have his or her public key. You encrypt with the recipient's public key, and only the recipient can decrypt with his or her private key (you can, of course, use your own public key if you're encrypting something for yourself). The first step in this process is importing the recipient's public key into your keyring. If you want to practice this, you can save my pubic key in a file called *oli.pub* and import it:

```
$ gpg --import oli.pub
```

List the public keys in your keyring:

```
$ gpg --list-keys
```

This will return something like:

```
pub   4096R/9R092F51 2014-03-20
uid        [ unknown] Jon Smith  <jonsmith@example.com>
sub   4096R/320G5188 2014-03-20
```

You can also list your non-public keys:

```
$ gpg --list-secret-keys
```

Your keys are stored in the folder:

```
~/.gnupg
```

To encrypt the file *tmp.txt* for the person with uid (user id) *Jon Smith* (where *Jon Smith*'s public key has already been imported into your keyring):

```
$ gpg --recipient "Jon Smith" --encrypt tmp.txt
```

(you can use the email address as well as the name)

This will create the file *tmp.txt.gpg*, which is in non-human-readable binary OpenPGP format. If you're encrypting a bit of text, you can use the *-a* flag, which creates "ASCII armored output"—i.e., a file you can `cat` or paste into the body of an email:

```
$ gpg --recipient "Jon Smith" --encrypt -a tmp.txt
```

This will create the file *tmp.txt.asc*, which will look something like this:

```
-----BEGIN PGP MESSAGE-----
Version: GnuPG v2

hQIMA4P86BWIDBBVAQ/+KIfnVGBF0Ei/9G/a8btxHu1wXrOpxp77ofPLzJ47e3pm
Y4uO17sbR9JJ12gPHoc5wQT60De9JNtSsPbyD7HVUF0kD+mfnyM8UlyWd7P4BjE5
vRZLhlMt4R88ZvjsU5yTRmLFSkMTIl5NUXTiGuBfjkRZUL+1FooUKgrpu5osACy/
6/7FZ75jReR8g/HEYHody4t8mA3bB5uLoZ8ZEHluj6hf6HjI8nFivNO487IHMhz3
UnDeoStL4lrx/zU0Depv/QNb4FRvOWUQMR7Mf61RcFtcHqfDyjg3Sh5/zg5icAc6
/GEx/6fIuQtmVlXtDCuS17XjaTNBdBXgkBqxsDwk3G1Ilgl9Jo83kBkQLgo3FdB6
3qs9AafNxTzloba8nF38fp0LuvvtSyhfHpYIc4URD5Mpt5ojIHNBVxevY0OeRDX8
x6Bmqu1tKSsKJz7fb0z3K/4/dYTMspIMUIw64y6W3De5jJv9pkbaQ/T1+y5oqOeD
rNvkMYsOpMvHBrnf0liMgn+sLuKE+G26lQfGm4MdnFQmb7AWBC5eqK8p1MnSojGm
klTlTpRSKfx4vOhB4K64L2hlH0rBHE3CvOOsJivzZ7r7MKsBoX6ZHwxVR0YOh/5J
m0Fzk0iprP9vzv5bWlADpCWYVUp6I6WHDfaFnwCDxH2O6y+krxHGjHei7u7GV9fS
SgEVvZZDErHr/ZTwwa7Xld37cJ9dOpesBECrk5ncLr25mNzDNGxgDXqM2yEuzhNa
HDmO0dVloPnVuQ/2SYL/4JP4Z6Uitm13nKQK
=55in
-----END PGP MESSAGE-----
```

Conversely, if somebody wants to send you a secret message or file, they're going to need your public key. You can export it and send it to them:

```
$ gpg --export -a "myusername" > mypublickey.txt
```

where *myusername* was the user name you chose during key generation. (There are also keyservers, such as keyserver.ubuntu.com, which host public keys.)

If somebody sends you the encrypted file *secret.txt.asc*, you can decrypt it with:

```
$ gpg --decrypt secret.txt.asc
```

Suppose you want to encrypt a whole folder rather than a single file. How can you accomplish this? In order to do this, simply tar your folder first then encrypt it:

```
$ tar -czvf myfolder.tar.gz myfolder
$ gpg --recipient "myusername" --encrypt myfolder.tar.gz
```

This will produce a file called *myfolder.tar.gz.gpg*.

Copying Your Keys to Another Computer

Suppose you want to transfer your keys to another computer or a hard drive. First, remind yourself what your keys are:

```
$ gpg --list-keys
$ gpg --list-secret-keys
```

Next, as outlined here, save them:

```
$ gpg --export -a "myusername" > my_public_key1.asc
$ gpg --export-secret-keys -a "myusername" > my_private_key1.asc
```

This is all you have to do if you're storing them on a hard drive. If you want to import them into a keyring on another computer, it's:

```
$ gpg --import my_public_key1.asc
$ gpg --import my_private_key1.asc
```

Deleting or Modifying Keys

Edit keys:

```
$ gpg --edit-key mykeyid
```

Delete a public key:

```
gpg --delete-key mykeyid
```

Delete a private key:

```
$ gpg --delete-secret-keys mykeyid
```

More on gpg

If you want to learn more, here are some good references:

Encrypting Files with gpg2

Zachary Voase: Openpgp for Complete Beginners

Alan Eliasen: GPG Tutorial

Ian Atkinson: Introduction to GnuPG

datamash TOP VIEW_AS_PAGE

Note: `datamash` is not a default shell program. You have to download and install it.

GNU `datamash` is a great program for crunching through text files and collapsing rows on a common ID or computing basic statistics. Here are some simple examples of what it can do.

Collapse rows in one column based on a common ID in another column:

```
$ cat file.txt
3       d
2       w
3       c
4       x
1       a
```

```
$ cat file.txt | datamash -g 1 collapse 2 -s -W
1       a
2       w
3       d,c
4       x
```

The *-g* flag is the ID column; the *collapse* field picks the second column; the *-s* flag pre-sorts the file; and the *-W* flag allows us to delimit on whitespace.

Average rows in one column on a common ID:

```
$ cat file.txt
A       1       3       SOME_OTHER_INFO
A       1       4       SOME_OTHER_INFO2
B       2       30      SOME_OTHER_INFO4
A       2       5       SOME_OTHER_INFO3
B       1       1       SOME_OTHER_INFO4
B       2       3       SOME_OTHER_INFO4
B       2       1       SOME_OTHER_INFO4
```

```
$ cat file.txt | datamash -s -g 1,2 mean 3 -f -s
A       1       3       SOME_OTHER_INFO         3.5
A       2       5       SOME_OTHER_INFO3        5
B       1       1       SOME_OTHER_INFO4        1
B       2       30      SOME_OTHER_INFO4        11.333333333333
```

In this case, the ID is the combination of columns one and two and the mean of column 3 is added as an additional column.

Simply sum a file of numbers:

```
$ cat file.txt | datamash sum 1
```

Hat tip: Albert

virtualenv TOP VIEW_AS_PAGE

Note: `virtualenv` is not a default shell program. You have to download and install it.

Do you use Python? `virtualenv` is a special command line tool for Python users. We learned about package managers in An Introduction to the Command-Line (on Unix-like systems) - Installing Programs on the Command Line, and Python's is called `pip`. Suppose you're working on a number of Python projects. One project has a number of dependencies and you've used `pip` to install them. Another project has a different set of dependencies, and so on. You could install all of your Python modules in your global copy of Python, but that could get messy. It would be nice if you could associate your dependencies with your particular project. This would also ensure that if two projects have conflicting dependencies—say they depend on different versions of the same module—you can get away with it. Moreover, it would allow you to freely install or update modules to your global Python worry-free, since this won't interfere with your projects. This is what `virtualenv` does and why it's a boon to Python users.

Following the docs, first install it:

```
$ sudo pip install virtualenv
```

To make a new Python installation in a folder called *venv*, run:

```
$ virtualenv venv
```

To emphasize the point, this is a whole new copy of Python. To use this Python, type:

```
$ source venv/bin/activate
```

As a sanity check, examine which Python you're using:

```
(venv) $ which python
/some/path/venv/bin/python
```

It's virtualenv's copy! Now if you, say, install Django:

```
(venv) $ pip install Django
```

You can see that you only have the Django module (and wheel):

```
(venv) $ pip freeze
Django==1.8.7
wheel==0.24.0
```

Django's source code is going to be installed in a path such as:

```
venv/lib/python2.7/site-packages
```

In practice, if you were doing a Django project, everytime you wanted to start coding, the first order of business would be to turn on virtualenv and the last would be to turn it off. To exit virtualenv, type:

```
(venv) $ deactivate
```

lsof TOP VIEW_AS_PAGE

`lsof` shows all open files on your computer and tells you which user and process ID is opening the particular file. This is surprisingly useful for debugging, among other things. Want to see what files *myprogram* is opening or writing to? Is it, say, writing to the `/tmp` directory? Try:

```
$ lsof | grep myprogram
```

To show your network connections, use:

```
$ lsof -i
```

Read more:
Wikipedia: lsof
Tecmint: 10 lsof Command Examples in Linux
IT'S ME, TOMMY: lsof

Bonus: Global Variables TOP VIEW_AS_PAGE

To see all global variables, type:

```
$ set
```

Where bash looks for commands and scripts:

```
$PATH
```

Add a directory path to the back of the path:

```
$ PATH=$PATH:/my/new/path
```

Add a directory path to the front of the path:

```
$ PATH=/my/new/path:$PATH
```

Now we know where to put bash commands, but what about other programs? What if you've installed a package or a module in a local directory and you want the program to have access to it? In that case, the following global variables come into play.

Where matlab looks for commands and scripts:

```
$MATLABPATH
```

Where R looks for packages:

```
$R_LIBS
```

Where awk looks for commands and scripts:

```
$AWKPATH
```

Where Python looks for modules:

```
$PYTHONPATH
```

Where Cpp looks for libraries:

```
$LD_LIBRARY_PATH
```

Where Perl looks for modules:

```
$PERL5LIB
```

This will come into play when you've locally installed a module and have to make sure Perl sees it. Then you'll probably have to `export` it. E.g.:

```
$ export PERL5LIB=/some/path/lib64/perl5
```

Text editor:

```
$EDITOR
```

You can set this by typing, e.g.:

```
export EDITOR=/usr/bin/nano
```

Then *Control-x-e* will invoke the text editor.

The specially designated temporary directory:

```
$TMPDIR
```

A shell variable that stores a random number for you:

```
$RANDOM
```

For example:

```
$ echo $RANDOM
1284
$ echo $RANDOM
27837
```

Bonus: Network Analysis TOP VIEW_AS_PAGE

Network sleuthing is far outside my expertise, but here are some utilities to check out:

`tshark`
`netstat`
`dig`
`nslookup`
`nmap`
`lsof`