Andrew (he/him)
Posted on 13 janv. 2019 • Updated on 25 sept. 2019

Linux Bash Commands

101 Bash Commands and Tips for Beginners to Experts

#beginners #bash #linux #ubuntu

> **Update 25 Sep 2019:** This article is now available in Japanese, thanks to the hard work of ラナ・クアール. Please check out their work by following the link below. If you're aware of translations of this article to other languages, please let me know and I'll post them here.

JP 日本語で読む

> **Update 8 July 2019:** I recently found this very similar article posted to a French-language message board about two years ago. If you're interested in learning some shell commands -- and you *parler français*, it's a great supplement to my article, below.

Until about a year ago, I worked almost exclusively within the macOS and Ubuntu operating systems. On both of those OSes, `bash` is my default shell. I've acquired a general understanding of how `bash` works over the past six or seven years and would like to give an overview of some of the more common / useful commands for those just getting started. If you think you know everything there is to know about `bash`, take a look below anyway -- I've included some tips and reminders of flags you may have forgotten about, which could make your work a bit easier.

The commands below are laid out in a more-or-less narrative style, so if you're just getting started with `bash`, you can work your way through from the beginning to the end. Things generally get less common and more difficult toward the end.

Table of Contents

The Basics

First Commands, Navigating the Filesystem

Modern filesystems have directory (folder) trees, where a directory is either a *root directory* (with no parent directory) or is a *subdirectory* (contained within a single other directory, which we call its "parent"). Traversing backwards through the file tree (from child directory to parent directory) will always get you to the root directory. Some filesystems have multiple root directories (like Windows' drives: `C:\`, `A:\`, etc.), but Unix and Unix-like systems only have a single root directory called `\`.

`pwd / ls / cd`

[Back to Table of Contents]

When working within a filesystem, the user is always working *within* some directory, which we call the current directory or the *working directory*. Print the user's working directory with `pwd`:

```
[ andrew@pc01 ~ ]$ pwd
/home/andrew
```

List the contents of this directory (files and/or child directories, etc.) with `ls`:

```
[ andrew@pc01 ~ ]$ ls
Git  TEST  jdoc  test  test.file
```

> **Bonus:**
>
> Show hidden ("dot") files with `ls -a`
>
> Show file details with `ls -l`
>
> Combine multiple flags like `ls -l -a`
>
> You can sometimes chain flags like `ls -la` instead of `ls -l -a`

Change to a different directory with `cd` (change directory):

```
[ andrew@pc01 ~ ]$ cd TEST/

[ andrew@pc01 TEST ]$ pwd
/home/andrew/TEST

[ andrew@pc01 TEST ]$ cd A
```

```
[ andrew@pc01 A ]$ pwd
/home/andrew/TEST/A
```

`cd ..` is shorthand for "`cd` to the parent directory":

```
[ andrew@pc01 A ]$ cd ..

[ andrew@pc01 TEST ]$ pwd
/home/andrew/TEST
```

`cd ~` or just `cd` is shorthand for "`cd` to my home directory" (usually `/home/username` or something similar):

```
[ andrew@pc01 TEST ]$ cd

[ andrew@pc01 ~ ]$ pwd
/home/andrew
```

> **Bonus:**
>
> `cd ~user` means "`cd` to `user`'s home directory
>
> You can jump multiple directory levels with `cd ../..`, etc.
>
> Go back to the most recent directory with `cd -`
>
> `.` is shorthand for "this directory", so `cd .` won't do much of anything

`;` / `&&` / `&`

[Back to Table of Contents]

The things we type into the command line are called *commands*, and they always execute some machine code stored somewhere on your computer. Sometimes this machine code is a built-in Linux command, sometimes it's an app, sometimes it's some code that you wrote yourself. Occasionally, we'll want to run one command right after another. To do that, we can use the `;` (semicolon):

```
[ andrew@pc01 ~ ]$ ls; pwd
Git  TEST  jdoc  test  test.file
/home/andrew
```

Above, the semicolon means that I first (`ls`) list the contents of the working directory, and then I (`pwd`) print its location. Another useful tool for chaining commands is `&&`.

With `&&`, the command to the right will not run if the command to the left fails. `;` and `&&` can both be used multiple times on the same line:

```
# whoops! I made a typo here!
[ andrew@pc01 ~ ]$ cd /Giit/Parser && pwd && ls && cd
-bash: cd: /Giit/Parser: No such file or directory

# the first command passes now, so the following commands are run
[ andrew@pc01 ~ ]$ cd Git/Parser/ && pwd && ls && cd
/home/andrew/Git/Parser
README.md  doc.sh  pom.xml  resource  run.sh  shell.sh  source  src  target
```

...but with `;`, the second command will run even if the first one fails:

```
# pwd and ls still run, even though the cd command failed
[ andrew@pc01 ~ ]$ cd /Giit/Parser ; pwd ; ls
-bash: cd: /Giit/Parser: No such file or directory
/home/andrew
Git  TEST  jdoc  test  test.file
```

`&` looks similar to `&&` but actually fulfils a completely different function. Normally, when you execute a long-running command, the command line will wait for that command to finish before it allows you to enter another one. Putting `&` after a command prevents this from happening, and lets you execute a new command while an older one is still going:

```
[ andrew@pc01 ~ ]$ cd Git/Parser && mvn package & cd
[1] 9263
```

> **Bonus:** When we use `&` after a command to "hide" it, we say that the job (or the "process"; these terms are more or less interchangeable) is "backgrounded". To see what background jobs are currently running, use the `jobs` command:
>
> ```
> [ andrew@pc01 ~ ]$ jobs
> [1]+ Running cd Git/Parser/ && mvn package &
> ```

Getting Help

`-h`

[Back to Table of Contents]

Type `-h` or `--help` after almost any command to bring up a help menu for that command:

```
[ andrew@pc01 ~ ]$ du --help
Usage: du [OPTION]... [FILE]...
  or:  du [OPTION]... --files0-from=F
Summarize disk usage of the set of FILEs, recursively for directories.

Mandatory arguments to long options are mandatory for short options too.
  -0, --null            end each output line with NUL, not newline
  -a, --all             write counts for all files, not just directories
      --apparent-size   print apparent sizes, rather than disk usage; although
                          the apparent size is usually smaller, it may be
                          larger due to holes in ('sparse') files, internal
                          fragmentation, indirect blocks, and the like
  -B, --block-size=SIZE  scale sizes by SIZE before printing them; e.g.,
                           '-BM' prints sizes in units of 1,048,576 bytes;
                           see SIZE format below
...
```

man

[Back to Table of Contents]

Type `man` before almost any command to bring up a manual for that command (quit `man` with `q`):

```
LS(1)                          User Commands                          LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information  about  the FILEs (the current directory by default).
       Sort entries alphabetically if none of -cftuvSUX nor --sort  is  speci-
       fied.

       Mandatory  arguments  to  long  options are mandatory for short options
       too.
...
```

Viewing and Editing Files

head / tail / cat / less

[Back to Table of Contents]

`head` outputs the first few lines of a file. The `-n` flag specifies the number of lines to show (the default is 10):

```
# prints the first three lines
[ andrew@pc01 ~ ]$ head -n 3 c
this
file
has
```

`tail` outputs the last few lines of a file. You can get the last `n` lines (like above), or you can get the end of the file beginning from the `N`-th line with `tail -n +N`:

```
# prints the end of the file, beginning with the 4th line
[ andrew@pc01 ~ ]$ tail -n +4 c
exactly
six
lines
```

`cat` concatenates a list of files and sends them to the standard output stream (usually the terminal). `cat` can be used with just a single file, or multiple files, and is often used to quickly view them. (**Be warned**: if you use `cat` in this way, you may be accused of a *Useless Use of Cat* (UUOC), but it's not that big of a deal, so don't worry too much about it.)

```
[ andrew@pc01 ~ ]$ cat a
file a

[ andrew@pc01 ~ ]$ cat a b
file a
file b
```

`less` is another tool for quickly viewing a file -- it opens up a `vim`-like read-only window. (Yes, there is a command called `more`, but `less` -- unintuitively -- offers a superset of the functionality of `more` and is recommended over it.) Learn more (or less?) about less and more at their `man` pages.

nano / nedit

[Back to Table of Contents]

`nano` is a minimalistic command-line text editor. It's a great editor for beginners or people who don't want to learn a million shortcuts. It was more than sufficient for me for the first few years of my coding career (I'm only now starting to look into more powerful editors, mainly because defining your own syntax highlighting in `nano` can be a bit of a pain.)

`nedit` is a small graphical editor, it opens up an X Window and allows point-and-click editing, drag-and-drop, syntax highlighting and more. I use `nedit` sometimes when I want to make small changes to a script and re-run it over and over.

Other common CLI (command-line interface) / GUI (graphical user interface) editors include `emacs`, `vi`, `vim`, `gedit`, Notepad++, Atom, and lots more. Some cool ones that I've played around with (and can endorse) include Micro, Light Table, and VS Code.

All modern editors offer basic conveniences like search and replace, syntax highlighting, and so on. `vi(m)` and `emacs` have more features than `nano` and `nedit`, but they have a much steeper learning curve. Try a few different editors out and find one that works for you!

Creating and Deleting Files and Directories

`touch`

[Back to Table of Contents]

`touch` was created to modify file timestamps, but it can also be used to quickly create an empty file. You can create a new file by opening it with a text editor, like `nano`:

```
[ andrew@pc01 ex ]$ ls

[ andrew@pc01 ex ]$ nano a
```

...editing file...

```
[ andrew@pc01 ex ]$ ls
a
```

...or by simply using `touch`:

```
[ andrew@pc01 ex ]$ touch b && ls
a  b
```

> **Bonus**:
>
> Background a process with ^z (Ctrl+z)

```
[ andrew@pc01 ex ]$ nano a
```

...editing file, then hit ^z...

```
Use fg to return to nano

[1]+ Stopped nano a
[ andrew@pc01 ex ]$ fg
```

...editing file again...

> **Double Bonus:**
>
> Kill the current (foreground) process by pressing ^c (Ctrl+c) while it's running
>
> Kill a background process with `kill %N` where `N` is the job index shown by the `jobs` command

`mkdir / rm / rmdir`

[Back to Table of Contents]

`mkdir` is used to create new, empty directories:

```
[ andrew@pc01 ex ]$ ls && mkdir c && ls
a  b
a  b  c
```

You can remove any file with `rm` -- but be careful, this is non-recoverable!

```
[ andrew@pc01 ex ]$ rm a && ls
b  c
```

You can add an "*are you sure?*" prompt with the `-i` flag:

```
[ andrew@pc01 ex ]$ rm -i b
rm: remove regular empty file 'b'? y
```

Remove an empty directory with `rmdir`. If you `ls -a` in an empty directory, you should only see a reference to the directory itself (`.`) and a reference to its parent directory (`..`):

```
[ andrew@pc01 ex ]$ rmdir c && ls -a
.  ..
```

`rmdir` removes empty directories only:

```
[ andrew@pc01 ex ]$ cd .. && ls test/
*.txt  0.txt  1.txt  a  a.txt  b  c

[ andrew@pc01 ~ ]$ rmdir test/
rmdir: failed to remove 'test/': Directory not empty
```

...but you can remove a directory -- and all of its contents -- with `rm -rf` (`-r` = recursive, `-f` = force):

```
[ andrew@pc01 ~ ]$ rm -rf test
```

Moving and Copying Files, Making Links, Command History

`mv / cp / ln`

[Back to Table of Contents]

`mv` moves / renames a file. You can `mv` a file to a new directory and keep the same file name or `mv` a file to a "new file" (rename it):

```
[ andrew@pc01 ex ]$ ls && mv a e && ls
a  b  c  d
b  c  d  e
```

`cp` copies a file:

```
[ andrew@pc01 ex ]$ cp e e2 && ls
b  c  d  e  e2
```

`ln` creates a hard link to a file:

```
# first argument to ln is TARGET, second is NEW LINK
[ andrew@pc01 ex ]$ ln b f && ls
b  c  d  e  e2  f
```

`ln -s` creates a soft link to a file:

```
[ andrew@pc01 ex ]$ ln -s b g && ls
b  c  d  e  e2  f  g
```

Hard links reference the same actual bytes in memory which contain a file, while soft links refer to the original file name, which itself points to those bytes. [You can read more about soft vs. hard links here.]

Command History

[Back to Table of Contents]

`bash` has two big features to help you complete and re-run commands, the first is *tab completion*. Simply type the first part of a command, hit the <tab> key, and let the terminal guess what you're trying to do:

```
[ andrew@pc01 dir ]$ ls <ENTER>
anotherlongfilename  thisisalongfilename  anewfilename

[ andrew@pc01 dir ]$ ls t <TAB>
```

...hit the TAB key after typing `ls t` and the command is completed...

```
[ andrew@pc01 dir ]$ ls thisisalongfilename <ENTER>
thisisalongfilename
```

You may have to hit <TAB> multiple times if there's an ambiguity:

```
[ andrew@pc01 dir ]$ ls a <TAB>

[ andrew@pc01 dir ]$ ls an <TAB>
anewfilename  anotherlongfilename
```

`bash` keeps a short history of the commands you've typed previously and lets you search through those commands by typing ^r (Ctrl+r):

```
[ andrew@pc01 dir ]
```

...hit ^r (Ctrl+r) to search the command history...

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

...type 'anew' and the last command containing this is found...

```
(reverse-i-search)`anew': touch anewfilename
```

Directory Trees, Disk Usage, and Processes

`mkdir -p / tree`

[Back to Table of Contents]

`mkdir`, by default, only makes a single directory. This means that if, for instance, directory `d/e` doesn't exist, then `d/e/f` can't be made with `mkdir` by itself:

```
[ andrew@pc01 ex ]$ ls && mkdir d/e/f
a  b  c
mkdir: cannot create directory 'd/e/f': No such file or directory
```

But if we pass the `-p` flag to `mkdir`, it will make all directories in the path if they don't already exist:

```
[ andrew@pc01 ex ]$ mkdir -p d/e/f && ls
a  b  c  d
```

`tree` can help you better visualise a directory's structure by printing a nicely-formatted directory tree. By default, it prints the entire tree structure (beginning with the specified directory), but you can restrict it to a certain number of levels with the `-L` flag:

```
[ andrew@pc01 ex ]$ tree -L 2
.
|-- a
|-- b
|-- c
`-- d
    `--e

3 directories, 2 files
```

You can hide empty directories in `tree`'s output with `--prune`. Note that this also removes "recursively empty" directories, or directories which aren't empty *per se*, but

which contain only other empty directories, or other recursively empty directories:

```
[ andrew@pc01 ex ]$ tree --prune
.
|-- a
`-- b
```

df / du / ps

[Back to Table of Contents]

`df` is used to show how much space is taken up by files for the disks or your system (hard drives, etc.).

```
[ andrew@pc01 ex ]$ df -h
Filesystem                   Size  Used Avail Use% Mounted on
udev                         126G     0  126G   0% /dev
tmpfs                         26G  2.0G   24G   8% /run
/dev/mapper/ubuntu--vg-root  1.6T  1.3T  252G  84% /
...
```

In the above command, `-h` doesn't mean "help", but "human-readable". Some commands use this convention to display file / disk sizes with `K` for kilobytes, `G` for gigabytes, and so on, instead of writing out a gigantic integer number of bytes.

`du` shows file space usage for a particular directory and its subdirectories. If you want to know how much space is free on a given hard drive, use `df`; if you want to know how much space a directory is taking up, use `du`:

```
[ andrew@pc01 ex ]$ du
4       ./d/e/f
8       ./d/e
12      ./d
4       ./c
20      .
```

`du` takes a `--max-depth=N` flag, which only shows directories `N` levels down (or fewer) from the specified directory:

```
[ andrew@pc01 ex ]$ du -h --max-depth=1
12K     ./d
4.0K    ./c
20K     .
```

`ps` shows all of the user's currently-running processes (aka. jobs):

```
[ andrew@pc01 ex ]$ ps
  PID TTY          TIME CMD
16642 pts/15   00:00:00 ps
25409 pts/15   00:00:00 bash
```

Miscellaneous

passwd / logout / exit

[Back to Table of Contents]

Change your account password with `passwd`. It will ask for your current password for verification, then ask you to enter the new password twice, so you don't make any typos:

```
[ andrew@pc01 dir ]$ passwd
Changing password for andrew.
(current) UNIX password:    <type current password>
Enter new UNIX password:    <type new password>
Retype new UNIX password:   <type new password again>
passwd: password updated successfully
```

`logout` exits a shell you've logged in to (where you have a user account):

```
[ andrew@pc01 dir ]$ logout
```

```
Session stopped
    - Press <return> to exit tab
    - Press R to restart session
    - Press S to save terminal output to file
```

`exit` exits any kind of shell:

```
[ andrew@pc01 ~ ]$ exit
logout
```

```
Session stopped
    - Press <return> to exit tab
    - Press R to restart session
    - Press S to save terminal output to file
```

`clear / *`

[Back to Table of Contents]

Run `clear` to move the current terminal line to the top of the screen. This command just adds blank lines below the current prompt line. It's good for clearing your workspace.

Use the glob (`*`, aka. Kleene Star, aka. wildcard) when looking for files. Notice the difference between the following two commands:

```
[ andrew@pc01 ~ ]$ ls Git/Parser/source/
PArrayUtils.java     PFile.java            PSQLFile.java      PWatchman.java
PDateTimeUtils.java  PFixedWidthFile.java  PStringUtils.java  PXSVFile.java
PDelimitedFile.java  PNode.java            PTextFile.java     Parser.java

[ andrew@pc01 ~ ]$ ls Git/Parser/source/PD*
Git/Parser/source/PDateTimeUtils.java  Git/Parser/source/PDelimitedFile.java
```

The glob can be used multiple times in a command and matches zero or more characers:

```
[ andrew@pc01 ~ ]$ ls Git/Parser/source/P*D*m*
Git/Parser/source/PDateTimeUtils.java  Git/Parser/source/PDelimitedFile.java
```

Intermediate

Disk, Memory, and Processor Usage

`ncdu`

[Back to Table of Contents]

`ncdu` (NCurses Disk Usage) provides a navigable overview of file space usage, like an improved `du`. It opens a read-only `vim`-like window (press `q` to quit):

```
[ andrew@pc01 ~ ]$ ncdu

ncdu 1.11 ~ Use the arrow keys to navigate, press ? for help
--- /home/andrew ------------------------------------------------------------
  148.2 MiB [##########] /.m2
   91.5 MiB [######    ] /.sbt
   79.8 MiB [#####     ] /.cache
   64.9 MiB [####      ] /.ivy2
```

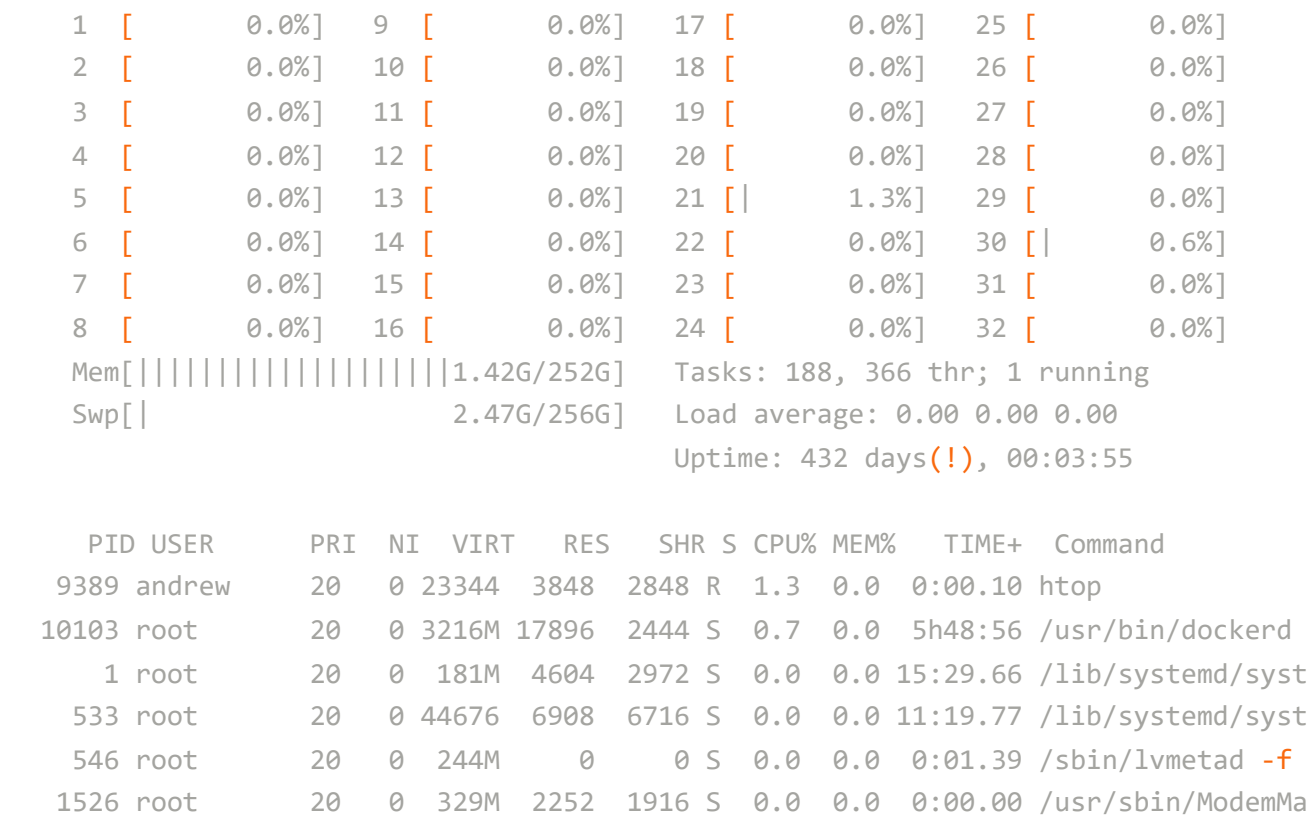
```
  40.6 MiB [##        ] /.sdkman
  30.2 MiB [##        ] /.local
  27.4 MiB [#         ] /.mozilla
  24.4 MiB [#         ] /.nanobackups
  10.2 MiB [          ]  .confout3.txt
   8.4 MiB [          ] /.config
   5.9 MiB [          ] /.nbi
   5.8 MiB [          ] /.oh-my-zsh
   4.3 MiB [          ] /Git
   3.7 MiB [          ] /.myshell
   1.7 MiB [          ] /jdoc
   1.5 MiB [          ]  .confout2.txt
   1.5 MiB [          ] /.netbeans
   1.1 MiB [          ] /.jenv
 564.0 KiB [          ] /.rstudio-desktop
Total disk usage: 552.7 MiB  Apparent size: 523.6 MiB  Items: 14618
```

top / htop

[Back to Table of Contents]

`top` displays all currently-running processes and their owners, memory usage, and more. `htop` is an improved, interactive `top`. (Note: you can pass the `-u username` flag to restrict the displayed processes to only those owner by `username`.)

```
[ andrew@pc01 ~ ]$ htop
```



```
 1544 root       20   0  329M  2252  1916 S  0.0  0.0  0:00.06 /usr/sbin/ModemMa
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

REPLs and Software Versions

REPLs

[Back to Table of Contents]

A **REPL** is a Read-Evaluate-Print Loop, similar to the command line, but usually used for particular programming languages.

You can open the Python REPL with the `python` command (and quit with the `quit()` function):

```
[ andrew@pc01 ~ ]$ python
Python 3.5.2 (default, Nov 12 2018, 13:43:14) ...
>>> quit()
```

Open the R REPL with the `R` command (and quit with the `q()` function):

```
[ andrew@pc01 ~ ]$ R
R version 3.5.2 (2018-12-20) --"Eggshell Igloo" ...
> q()
Save workspace image? [y/n/c]: n
```

Open the Scala REPL with the `scala` command (and quit with the `:quit` command):

```
[ andrew@pc01 ~ ]$ scala
Welcome to Scala 2.11.12 ...
scala> :quit
```

Open the Java REPL with the `jshell` command (and quit with the `/exit` command):

```
[ andrew@pc01 ~ ]$ jshell
| Welcome to JShell--Version 11.0.1 ...
jshell> /exit
```

Alternatively, you can exit any of these REPLs with ^d (Ctrl+d). ^d is the EOF (end of file) marker on Unix and signifies the end of input.

`-version / --version / -v`

[Back to Table of Contents]

Most commands and programs have a `-version` or `--version` flag which gives the software version of that command or program. Most applications make this information easily available:

```
[ andrew@pc01 ~ ]$ ls --version
ls (GNU coreutils) 8.25 ...

[ andrew@pc01 ~ ]$ ncdu -version
ncdu 1.11

[ andrew@pc01 ~ ]$ python --version
Python 3.5.2
```

...but some are less intuitive:

```
[ andrew@pc01 ~ ]$ sbt scalaVersion
...
[info] 2.12.4
```

Note that some programs use `-v` as a version flag, while others use `-v` to mean "verbose", which will run the application while printing lots of diagnostic or debugging information:

```
SCP(1)                    BSD General Commands Manual                    SCP(1)

NAME
     scp -- secure copy (remote file copy program)
...
-v      Verbose mode.  Causes scp and ssh(1) to print debugging messages
             about their progress.  This is helpful in debugging connection,
             authentication, and configuration problems.
...
```

Environment Variables and Aliases

Environment Variables

[Back to Table of Contents]

Environment variables (sometimes shortened to "env vars") are persistent variables that can be created and used within your `bash` shell. They are defined with an equals sign (`=`) and used with a dollar sign (`$`). You can see all currently-defined env vars with `printenv`:

```
[ andrew@pc01 ~ ]$ printenv
SPARK_HOME=/usr/local/spark
TERM=xterm
...
```

Set a new environment variable with an `=` sign (don't put any spaces before or after the `=`, though!):

```
[ andrew@pc01 ~ ]$ myvar=hello
```

Print a specific env var to the terminal with `echo` and a preceding `$` sign:

```
[ andrew@pc01 ~ ]$ echo $myvar
hello
```

Environment variables which contain spaces or other whitespace should be surrounded by quotes (`"..."`). Note that reassigning a value to an env var overwrites it without warning:

```
[ andrew@pc01 ~ ]$ myvar="hello, world!" && echo $myvar
hello, world!
```

Env vars can also be defined using the `export` command. When defined this way, they will also be available to sub-processes (commands called from this shell):

```
[ andrew@pc01 ~ ]$ export myvar="another one" && echo $myvar
another one
```

You can unset an environment variable by leaving the right-hand side of the `=` blank or by using the `unset` command:

```
[ andrew@pc01 ~ ]$ unset mynewvar

[ andrew@pc01 ~ ]$ echo $mynewvar
```

Aliases

[Back to Table of Contents]

Aliases are similar to environment variables but are usually used in a different way -- to replace long commands with shorter ones:

```
[ andrew@pc01 apidocs ]$ ls -l -a -h -t
total 220K
drwxr-xr-x 5 andrew andrew 4.0K Dec 21 12:37 .
-rw-r--r-- 1 andrew andrew 9.9K Dec 21 12:37 help-doc.html
-rw-r--r-- 1 andrew andrew 4.5K Dec 21 12:37 script.js
...

[ andrew@pc01 apidocs ]$ alias lc="ls -l -a -h -t"

[ andrew@pc01 apidocs ]$ lc
total 220K
drwxr-xr-x 5 andrew andrew 4.0K Dec 21 12:37 .
-rw-r--r-- 1 andrew andrew 9.9K Dec 21 12:37 help-doc.html
-rw-r--r-- 1 andrew andrew 4.5K Dec 21 12:37 script.js
...
```

You can remove an alias with `unalias`:

```
[ andrew@pc01 apidocs ]$ unalias lc

[ andrew@pc01 apidocs ]$ lc
The program 'lc' is currently not installed. ...
```

> **Bonus:**
>
> Read about the subtle differences between environment variables and aliases here.
>
> Some programs, like **git**, allow you to define aliases specifically for that software.

Basic `bash` Scripting

`bash` Scripts

[Back to Table of Contents]

`bash` scripts (usually ending in `.sh`) allow you to automate complicated processes, packaging them into reusable functions. A `bash` script can contain any number of normal shell commands:

```
[ andrew@pc01 ~ ]$ echo "ls && touch file && ls" > ex.sh
```

A shell script can be executed with the `source` command or the `sh` command:

```
[ andrew@pc01 ~ ]$ source ex.sh
Desktop  Git  TEST  c  ex.sh  project  test
```

```
Desktop  Git  TEST  c  ex.sh  file  project  test
```

Shell scripts can be made executable with the `chmod` command (more on this later):

```
[ andrew@pc01 ~ ]$ echo "ls && touch file2 && ls" > ex2.sh

[ andrew@pc01 ~ ]$ chmod +x ex2.sh
```

An executable shell script can be run by preceding it with `./`:

```
[ andrew@pc01 ~ ]$ ./ex2.sh
Desktop  Git  TEST  c  ex.sh  ex2.sh  file  project  test
Desktop  Git  TEST  c  ex.sh  ex2.sh  file  file2  project  test
```

Long lines of code can be split by ending a command with `\`:

```
[ andrew@pc01 ~ ]$ echo "for i in {1..3}; do echo \
> \"Welcome \$i times\"; done" > ex3.sh
```

Bash scripts can contain loops, functions, and more!

```
[ andrew@pc01 ~ ]$ source ex3.sh
Welcome 1 times
Welcome 2 times
Welcome 3 times
```

Custom Prompt and `ls`

[Back to Table of Contents]

Bash scripting can make your life a whole lot easier and more colourful. Check out this great bash scripting cheat sheet.

`$PS1` (Prompt String 1) is the environment variable that defines your main shell prompt (learn about the other prompts here):

```
[ andrew@pc01 ~ ]$ printf "%q" $PS1
$'\\n\\[\E[1m\\]\\[\E[30m\\]\\A'$'\\[\E[37m\\]|\\[\E[36m\\]\\u\\[\E[37m\\]@\\[\E[34m\\
```

You can change your default prompt with the `export` command:

```
[ andrew@pc01 ~ ]$ export PS1="\ncommand here> "
```

```
command here> echo $PS1
\ncommand here>
```

...you can add colours, too!:

```
command here> export PS1="\e[1;31m\nCODE: \e[39m"

# (this should be red, but it may not show up that way in Markdown)
CODE: echo $PS1
\e[1;31m\nCODE: \e[39m
```

You can also change the colours shown by `ls` by editing the `$LS_COLORS` environment variable:

```
# (again, these colours might not show up in Markdown)
CODE: ls
Desktop  Git  TEST  c  ex.sh  ex2.sh  ex3.sh  file  file2  project  test

CODE: export LS_COLORS='di=31:fi=0:ln=96:or=31:mi=31:ex=92‘

CODE: ls
Desktop  Git  TEST  c  ex.sh  ex2.sh  ex3.sh  file  file2  project  test
```

Config Files

Config Files / `.bashrc`

[Back to Table of Contents]

If you tried the commands in the last section and logged out and back in, you may have noticed that your changes disappeared. *config* (configuration) files let you maintain settings for your shell or for a particular program every time you log in (or run that program). The main configuration file for a `bash` shell is the `~/.bashrc` file. Aliases, environment variables, and functions added to `~/.bashrc` will be available every time you log in. Commands in `~/.bashrc` will be run every time you log in.

If you edit your `~/.bashrc` file, you can reload it without logging out by using the `source` command:

```
[ andrew@pc01 ~ ]$ nano ~/.bashrc
```

...add the line `echo “~/.bashrc loaded!”` to the top of the file...

```
[ andrew@pc01 ~ ]$ source ~/.bashrc
~/.bashrc loaded!
```

...log out and log back in...

```
Last login: Fri Jan 11 10:29:07 2019 from 111.11.11.111
~/.bashrc loaded!

[ andrew@pc01 ~ ]
```

Types of Shells

[Back to Table of Contents]

Login shells are shells you log in to (where you have a username). *Interactive* shells are shells which accept commands. Shells can be login and interactive, non-login and non-interactive, or any other combination.

In addition to `~/.bashrc`, there are a few other scripts which are `sourced` by the shell automatically when you log in or log out. These are:

- `/etc/profile`
- `~/.bash_profile`
- `~/.bash_login`
- `~/.profile`
- `~/.bash_logout`
- `/etc/bash.bash_logout`

Which of these scripts are sourced, and the order in which they're sourced, depend on the type of shell opened. See the bash man page and these Stack Overflow posts for more information.

Note that `bash` scripts can `source` other scripts. For instance, in your `~/.bashrc`, you could include the line:

```
source ~/.bashrc_addl
```

...which would also `source` that `.bashrc_addl` script. This file can contain its own aliases, functions, environment variables, and so on. It could, in turn, `source` other scripts, as well. (Be careful to avoid infinite loops of script-sourcing!)

It may be helpful to split commands into different shell scripts based on functionality or machine type (Ubuntu vs. Red Hat vs. macOS), for example:

- `~/.bash_ubuntu` -- configuration specific to Ubuntu-based machines
- `~/.bashrc_styles` -- aesthetic settings, like `PS1` and `LS_COLORS`
- `~/.bash_java` -- configuration specific to the Java language

I try to keep separate `bash` files for aesthetic configurations and OS- or machine-specific code, and then I have one big `bash` file containing shortcuts, etc. that I use on every machine and every OS.

Note that there are also *different shells*. `bash` is just one kind of shell (the "Bourne Again Shell"). Other common ones include `zsh`, `csh`, `fish`, and more. Play around with different shells and find one that's right for you, but be aware that this tutorial contains `bash` shell commands only and not everything listed here (maybe none of it) will be applicable to shells other than `bash`.

Finding Things

`whereis / which / whatis`

[Back to Table of Contents]

`whereis` searches for "possibly useful" files related to a particular command. It will attempt to return the location of the binary (executable machine code), source (code source files), and `man` page for that command:

```
[ andrew@pc01 ~ ]$ whereis ls
ls: /bin/ls /usr/share/man/man1/ls.1.gz
```

`which` will only return the location of the binary (the command itself):

```
[ andrew@pc01 ~ ]$ which ls
/bin/ls
```

`whatis` prints out the one-line description of a command from its `man` page:

```
[ andrew@pc01 ~ ]$ whatis whereis which whatis
whereis (1)          - locate the binary, source, and manual page files for a command
which (1)            - locate a command
whatis (1)           - display one-line manual page descriptions
```

`which` is useful for finding the "original version" of a command which may be hidden by an alias:

```
[ andrew@pc01 ~ ]$ alias ls="ls -l"

# “original” ls has been “hidden” by the alias defined above
[ andrew@pc01 ~ ]$ ls
total 36
drwxr-xr-x 2 andrew andrew 4096 Jan  9 14:47 Desktop
drwxr-xr-x 4 andrew andrew 4096 Dec  6 10:43 Git
...

# but we can still call “original” ls by using the location returned by which
[ andrew@pc01 ~ ]$ /bin/ls
Desktop  Git  TEST  c  ex.sh  ex2.sh  ex3.sh  file  file2  project  test
```

locate / find

[Back to Table of Contents]

`locate` finds a file anywhere on the system by referring to a semi-regularly-updated cached list of files:

```
[ andrew@pc01 ~ ]$ locate README.md
/home/andrew/.config/micro/plugins/gotham-colors/README.md
/home/andrew/.jenv/README.md
/home/andrew/.myshell/README.md
...
```

Because it's just searching a list, `locate` is usually faster than the alternative, `find`. `find` iterates through the file system to find the file you're looking for. Because it's actually looking at the files which *currently* exist on the system, though, it will always return an up-to-date list of files, which is not necessarily true with `locate`.

```
[ andrew@pc01 ~ ]$ find ~/ -iname "README.md"
/home/andrew/.jenv/README.md
/home/andrew/.config/micro/plugins/gotham-colors/README.md
/home/andrew/.oh-my-zsh/plugins/ant/README.md
...
```

`find` was written for the very first version of Unix in 1971, and is therefore much more widely available than `locate`, which was added to GNU in 1994.

`find` has many more features than `locate`, and can search by file age, size, ownership, type, timestamp, permissions, depth within the file system; `find` can search using regular expressions, execute commands on files it finds, and more.

When you need a fast (but possibly outdated) list of files, or you're not sure what directory a particular file is in, use `locate`. When you need an accurate file list, maybe based on something other than the files' names, and you need to do something with those files, use `find`.

Downloading Things

ping / wget / curl

[Back to Table of Contents]

`ping` attempts to open a line of communication with a network host. Mainly, it's used to check whether or not your Internet connection is down:

```
[ andrew@pc01 ~ ]$ ping google.com
PING google.com (74.125.193.100) 56(84) bytes of data.
Pinging 74.125.193.100 with 32 bytes of data:
Reply from 74.125.193.100: bytes=32 time<1ms TTL=64
...
```

`wget` is used to easily download a file from the Internet:

```
[ andrew@pc01 ~ ]$ wget \
> http://releases.ubuntu.com/18.10/ubuntu-18.10-desktop-amd64.iso
```

`curl` can be used just like `wget` (don't forget the `--output` flag):

```
[ andrew@pc01 ~ ]$ curl \
> http://releases.ubuntu.com/18.10/ubuntu-18.10-desktop-amd64.iso \
> --output ubuntu.iso
```

`curl` and `wget` have their own strengths and weaknesses. `curl` supports many more protocols and is more widely available than `wget`; `curl` can also send data, while `wget` can only receive data. `wget` can download files recursively, while `curl` cannot.

In general, I use `wget` when I need to download things from the Internet. I don't often need to send data using `curl`, but it's good to be aware of it for the rare occasion that you do.

apt / gunzip / tar / gzip

[Back to Table of Contents]

Debian-descended Linux distributions have a fantastic package management tool called `apt`. It can be used to install, upgrade, or delete software on your machine. To search `apt` for a particular piece of software, use `apt search`, and install it with `apt install`:

```
[ andrew@pc01 ~ ]$ apt search bleachbit
...bleachbit/bionic,bionic 2.0-2 all
  delete unnecessary files from the system

# you need to 'sudo' to install software
[ andrew@pc01 ~ ]$ sudo apt install bleachbit
```

Linux software often comes packaged in `.tar.gz` ("tarball") files:

```
[ andrew@pc01 ~ ]$ wget \
> https://github.com/atom/atom/releases/download/v1.35.0-beta0/atom-amd64.tar.gz
```

...these types of files can be unzipped with `gunzip`:

```
[ andrew@pc01 ~ ]$ gunzip atom-amd64.tar.gz && ls
atom-amd64.tar
```

A `.tar.gz` file will be `gunzip`-ped to a `.tar` file, which can be extracted to a directory of files using `tar -xf` (`-x` for "extract", `-f` to specify the file to "untar"):

```
[ andrew@pc01 ~ ]$ tar -xf atom-amd64.tar && mv \
atom-beta-1.35.0-beta0-amd64 atom && ls
atom atom-amd64.tar
```

To go in the reverse direction, you can create (`-c`) a tar file from a directory and zip it (or unzip it, as appropriate) with `-z`:

```
[ andrew@pc01 ~ ]$ tar -zcf compressed.tar.gz atom && ls
atom  atom-amd64.tar  compressed.tar.gz
```

`.tar` files can also be zipped with `gzip`:

```
[ andrew@pc01 ~ ]$ gzip atom-amd64.tar && ls
atom  atom-amd64.tar.gz compressed.tar.gz
```

Redirecting Input and Output

`|` / `>` / `<` / `echo` / `printf`

[Back to Table of Contents]

By default, shell commands read their input from the standard input stream (aka. stdin or 0) and write to the standard output stream (aka. stdout or 1), unless there's an error, which is written to the standard error stream (aka. stderr or 2).

`echo` writes text to stdout by default, which in most cases will simply print it to the terminal:

```
[ andrew@pc01 ~ ]$ echo "hello"
hello
```

The pipe operator, `|`, redirects the output of the first command to the input of the second command:

```
# 'wc' (word count) returns the number of lines, words, bytes in a file
[ andrew@pc01 ~ ]$ echo "example document" | wc
      1       2      17
```

`>` redirects output from stdout to a particular location

```
[ andrew@pc01 ~ ]$ echo "test" > file && head file
test
```

`printf` is an improved `echo`, allowing formatting and escape sequences:

```
[ andrew@pc01 ~ ]$ printf "1\n3\n2"
1
3
2
```

`<` gets input from a particular location, rather than stdin:

```
# 'sort' sorts the lines of a file alphabetically / numerically
[ andrew@pc01 ~ ]$ sort <(printf "1\n3\n2")
1
2
3
```

Rather than a UUOC, the recommended way to send the contents of a file to a command is to use `<`. Note that this causes data to "flow" right-to-left on the command line, rather than (the perhaps more natural, for English-speakers) left-to-right:

```
[ andrew@pc01 ~ ]$ printf "1\n3\n2" > file && sort < file
1
2
3
```

0 / 1 / 2 / tee

[Back to Table of Contents]

0, 1, and 2 are the standard input, output, and error streams, respectively. Input and output streams can be redirected with the `|`, `>`, and `<` operators mentioned previously, but stdin, stdout, and stderr can also be manipulated directly using their numeric identifiers:

Write to stdout or stderr with `>&1` or `>&2`:

```
[ andrew@pc01 ~ ]$ cat test
echo "stdout" >&1
echo "stderr" >&2
```

By default, stdout and stderr both print output to the terminal:

```
[ andrew@pc01 ~ ]$ ./test
stderr
stdout
```

Redirect stdout to `/dev/null` (only print output sent to stderr):

```
[ andrew@pc01 ~ ]$ ./test 1>/dev/null
stderr
```

Redirect stderr to `/dev/null` (only print output sent to stdout):

```
[ andrew@pc01 ~ ]$ ./test 2>/dev/null
stdout
```

Redirect all output to `/dev/null` (print nothing):

```
[ andrew@pc01 ~ ]$ ./test &>/dev/null
```

Send output to stdout and any number of additional locations with `tee`:

```
[ andrew@pc01 ~ ]$ ls && echo "test" | tee file1 file2 file3 && ls
file0
test
file0  file1  file2  file3
```

Advanced

Superuser

sudo / su

[Back to Table of Contents]

You can check what your username is with `whoami`:

```
[ andrew@pc01 abc ]$ whoami
andrew
```

...and run a command as another user with `sudo -u username` (you will need that user's password):

```
[ andrew@pc01 abc ]$ sudo -u test touch def && ls -l
total 0
-rw-r--r-- 1 test test 0 Jan 11 20:05 def
```

If `-u` is not provided, the default user is the superuser (usually called "root"), with unlimited permissions:

```
[ andrew@pc01 abc ]$ sudo touch ghi && ls -l
total 0
-rw-r--r-- 1 test test 0 Jan 11 20:05 def
-rw-r--r-- 1 root root 0 Jan 11 20:14 ghi
```

Use `su` to become another user temporarily (and `exit` to switch back):

```
[ andrew@pc01 abc ]$ su test
Password:
test@pc01:/home/andrew/abc$ whoami
test
```

```
test@pc01:/home/andrew/abc$ exit
exit

[ andrew@pc01 abc ]$ whoami
andrew
```

Learn more about the differences between `sudo` and `su` here.

!!

[Back to Table of Contents]

The superuser (usually "root") is the only person who can install software, create users, and so on. Sometimes it's easy to forget that, and you may get an error:

```
[ andrew@pc01 ~ ]$ apt install ruby
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13: Permission denied)
E: Unable to acquire the dpkg frontend lock (/var/lib/dpkg/lock-frontend), are you roo
```

You could retype the command and add `sudo` at the front of it (run it as the superuser):

```
[ andrew@pc01 ~ ]$ sudo apt install ruby
Reading package lists...
```

Or, you could use the `!!` shortcut, which retains the previous command:

```
[ andrew@pc01 ~ ]$ apt install ruby
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13: Permission denied)
E: Unable to acquire the dpkg frontend lock (/var/lib/dpkg/lock-frontend), are you roo

[ andrew@pc01 ~ ]$ sudo !!
sudo apt install ruby
Reading package lists...
```

By default, running a command with `sudo` (and correctly entering the password) allows the user to run superuser commands for the next 15 minutes. Once those 15 minutes are up, the user will again be prompted to enter the superuser password if they try to run a restricted command.

File Permissions

File Permissions

[Back to Table of Contents]

Files may be able to be read (`r`), written to (`w`), and/or executed (`x`) by different users or groups of users, or not at all. File permissions can be seen with the `ls -l` command and are represented by 10 characters:

```
[ andrew@pc01 ~ ]$ ls -lh
total 8
drwxr-xr-x 4 andrew andrew 4.0K Jan  4 19:37 tast
-rwxr-xr-x 1 andrew andrew   40 Jan 11 16:16 test
-rw-r--r-- 1 andrew andrew    0 Jan 11 16:34 tist
```

The first character of each line represents the type of file, (`d` = directory, `l` = link, `-` = regular file, and so on); then there are three groups of three characters which represent the permissions held by the user (u) who owns the file, the permissions held by the group (g) which owns the file, and the permissions held any other (o) users. (The number which follows this string of characters is the number of links in the file system to that file (4 or 1 above).)

`r` means that person / those people have read permission, `w` is write permission, `x` is execute permission. If a directory is "executable", that means it can be opened and its contents can be listed. These three permissions are often represented with a single three-digit number, where, if `x` is enabled, the number is incremented by 1, if `w` is enabled, the number is incremented by 2, and if `r` is enabled, the number is incremented by 4. Note that these are equivalent to binary digits (`r-x -> 101 -> 5`, for example). So the above three files have permissions of 755, 755, and 644, respectively.

The next two strings in each list are the name of the owner (`andrew`, in this case) and the group of the owner (also `andrew`, in this case). Then comes the size of the file, its most recent modification time, and its name. The `-h` flag makes the output human readable (i.e. printing `4.0K` instead of `4096` bytes).

chmod / chown

[Back to Table of Contents]

File permissions can be modified with `chmod` by setting the access bits:

```
[ andrew@pc01 ~ ]$ chmod 777 test && chmod 000 tist && ls -lh
total 8.0K
drwxr-xr-x 4 andrew andrew 4.0K Jan  4 19:37 tast
-rwxrwxrwx 1 andrew andrew   40 Jan 11 16:16 test
---------- 1 andrew andrew    0 Jan 11 16:34 tist
```

...or by adding (`+`) or removing (`-`) `r`, `w`, and `x` permissions with flags:

```
[ andrew@pc01 ~ ]$ chmod +rwx tist && chmod -w test && ls -lh
chmod: test: new permissions are r-xrwxrwx, not r-xr-xr-x
total 8.0K
drwxr-xr-x 4 andrew andrew 4.0K Jan  4 19:37 tast
-r-xrwxrwx 1 andrew andrew   40 Jan 11 16:16 test
-rwxr-xr-x 1 andrew andrew    0 Jan 11 16:34 tist
```

The user who owns a file can be changed with `chown`:

```
[ andrew@pc01 ~ ]$ sudo chown marina test
```

The group which owns a file can be changed with `chgrp`:

```
[ andrew@pc01 ~ ]$ sudo chgrp hadoop tist && ls -lh
total 8.0K
drwxr-xr-x 4 andrew andrew 4.0K Jan  4 19:37 tast
-----w--w- 1 marina andrew   40 Jan 11 16:16 test
-rwxr-xr-x 1 andrew hadoop    0 Jan 11 16:34 tist
```

User and Group Management

Users

[Back to Table of Contents]

`users` shows all users currently logged in. Note that a user can be logged in multiple times if -- for instance -- they're connected via multiple `ssh` sessions.

```
[ andrew@pc01 ~ ]$ users
andrew colin colin colin colin colin krishna krishna
```

To see all users (even those not logged in), check `/etc/passwd`. (**WARNING**: do not modify this file! You can corrupt your user accounts and make it impossible to log in to your system.)

```
[ andrew@pc01 ~ ]$ alias au="cut -d: -f1 /etc/passwd \
> | sort | uniq" && au
 _apt
anaid
andrew...
```

Add a user with `useradd`:

```
[ andrew@pc01 ~ ]$ sudo useradd aardvark && au
_apt
aardvark
anaid...
```

Delete a user with `userdel`:

```
[ andrew@pc01 ~ ]$ sudo userdel aardvark && au
_apt
anaid
andrew...
```

Change a user's default shell, username, password, or group membership with `usermod`.

Groups

[Back to Table of Contents]

`groups` shows all of the groups of which the current user is a member:

```
[ andrew@pc01 ~ ]$ groups
andrew adm cdrom sudo dip plugdev lpadmin sambashare hadoop
```

To see all groups on the system, check `/etc/group`. (**DO NOT MODIFY** this file unless you know what you are doing.)

```
[ andrew@pc01 ~ ]$ alias ag="cut -d: -f1 /etc/group \
> | sort" && ag
adm
anaid
andrew...
```

Add a group with `groupadd`:

```
[ andrew@pc01 ~ ]$ sudo groupadd aardvark && ag
aardvark
adm
anaid...
```

Delete a group with `groupdel`:

```
[ andrew@pc01 ~ ]$ sudo groupdel aardvark && ag
adm
anaid
andrew...
```

Change a group's name, ID number, or password with `groupmod`.

Text Processing

uniq / sort / diff / cmp

[Back to Table of Contents]

`uniq` can print unique lines (default) or repeated lines:

```
[ andrew@pc01 man ]$ printf "1\n2\n2" > a && \> printf "1\n3\n2" > b

[ andrew@pc01 man ]$ uniq a
1
2
```

`sort` will sort lines alphabetically / numerically:

```
[ andrew@pc01 man ]$ sort b
1
2
3
```

`diff` will report which lines differ between two files:

```
[ andrew@pc01 man ]$ diff a b
2c2
< 2
---
> 3
```

`cmp` reports which bytes differ between two files:

```
[ andrew@pc01 man ]$ cmp a b
a b differ: char 3, line 2
```

cut / sed

[Back to Table of Contents]

`cut` is usually used to cut a line into sections on some delimiter (good for CSV processing). `-d` specifies the delimiter and `-f` specifies the field index to print (starting with 1 for the first field):

```
[ andrew@pc01 man ]$ printf "137.99.234.23" > c

[ andrew@pc01 man ]$ cut -d'.' c -f1
137
```

`sed` is commonly used to replace a string with another string in a file:

```
[ andrew@pc01 man ]$ echo "old" | sed s/old/new/
new
```

...but `sed` is an extremely powerful utility, and cannot be properly summarised here. It's actually Turing-complete, so it can do anything that any other programming language can do. `sed` can find and replace based on regular expressions, selectively print lines of a file which match or contain a certain pattern, edit text files in-place and non-interactively, and much more.

A few good tutorials on `sed` include:

- https://www.tutorialspoint.com/sed/
- http://www.grymoire.com/Unix/Sed.html
- https://www.computerhope.com/unix/used.htm

Pattern Matching

grep

[Back to Table of Contents]

The name `grep` comes from `g`/`re`/`p` (search `g`lobally for a `r`egular `e`xpression and `p`rint it); it's used for finding text in files.

`grep` is used to find lines of a file which match some pattern:

```
[ andrew@pc01 ~ ]$ grep -e ".*fi.*" /etc/profile
# /etc/profile: system-wide .profile file for the Bourne shell (sh(1))
    # The file bash.bashrc already sets the default PS1.
    fi
    fi
...
```

...or contain some word:

```
[ andrew@pc01 ~ ]$ grep "andrew" /etc/passwd
andrew:x:1000:1000:andrew,,,:/home/andrew:/bin/bash
```

`grep` is usually the go-to choice for simply finding matching lines in a file, if you're planning on allowing some other program to handle those lines (or if you just want to view them).

`grep` allows for (`-E`) use of extended regular expressions, (`-F`) matching any one of multiple strings at once, and (`-r`) recursively searching files within a directory. These flags used to be implemented as separate commands (`egrep`, `fgrep`, and `rgrep`, respectively), but those commands are now deprecated.

> **Bonus**: see the origins of the names of a few famous `bash` commands

awk

[Back to Table of Contents]

`awk` is a pattern-matching language built around reading and manipulating delimited data files, like CSV files.

As a rule of thumb, `grep` is good for finding strings and patterns in files, `sed` is good for one-to-one replacement of strings in files, and `awk` is good for extracting strings and patterns from files and analysing them.

As an example of what `awk` can do, here's a file containing two columns of data:

```
[ andrew@pc01 ~ ]$ printf "A 10\nB 20\nC 60" > file
```

Loop over the lines, add the number to sum, increment count, print the average:

```
[ andrew@pc01 ~ ]$ awk 'BEGIN {sum=0; count=0; OFS=" "} {sum+=$2; count++} END {print
Average: 30
```

`sed` and `awk` are both Turing-complete languages. There have been multiple books written about each of them. They can be extremely useful with pattern matching and text processing. I really don't have enough space here to do either of them justice. Go read more about them!

> **Bonus**: learn about some of the differences between `sed`, `grep`, and `awk`

Copying Files Over `ssh`

`ssh / scp`

[Back to Table of Contents]

`ssh` is how Unix-based machines connect to each other over a network:

```
[ andrew@pc01 ~ ]$ ssh -p <port> andrew@137.xxx.xxx.89
Last login: Fri Jan 11 12:30:52 2019 from 137.xxx.xxx.199
```

Notice that my prompt has changed as I'm now on a different machine:

```
[ andrew@pc02 ~ ]$ exit
logout
Connection to 137.xxx.xxx.89 closed.
```

Create a file on machine 1:

```
[ andrew@pc01 ~ ]$ echo "hello" > hello
```

Copy it to machine 2 using `scp` (secure copy; note that `scp` uses `-P` for a port #, `ssh` uses `-p`)

```
[ andrew@pc01 ~ ]$ scp -P <port> hello andrew@137.xxx.xxx.89:~
hello                                         100%    0     0.0KB/s   00:00
```

`ssh` into machine 2:

```
[ andrew@pc02 ~ ]$ ssh -p <port> andrew@137.xxx.xxx.89
Last login: Fri Jan 11 22:47:37 2019 from 137.xxx.xxx.79
```

The file's there!

```
[ andrew@pc02 ~ ]$ ls
hello  multi  xargs

[ andrew@pc02 ~ ]$ cat hello
hello
```

`rsync`

[Back to Table of Contents]

`rsync` is a file-copying tool which minimises the amount of data copied by looking for deltas (changes) between files.

Suppose we have two directories: `d`, with one file, and `s`, with two files:

```
[ andrew@pc01 d ]$ ls && ls ../s
f0
f0  f1
```

Sync the directories (copying only missing data) with `rsync`:

```
[ andrew@pc01 d ]$ rsync -av ../s/* .
sending incremental file list...
```

`d` now contains all files that `s` contains:

```
[ andrew@pc01 d ]$ ls
f0  f1
```

`rsync` can be performed over `ssh` as well:

```
[ andrew@pc02 r ]$ ls

[ andrew@pc02 r ]$ rsync -avz -e "ssh -p <port>" andrew@137.xxx.xxx.79:~/s/* .
receiving incremental file list
f0
f1

sent 62 bytes  received 150 bytes  141.33 bytes/sec
total size is 0  speedup is 0.00

[ andrew@pc02 r ]$ ls
f0  f1
```

Long-Running Processes

`yes / nohup / ps / kill`

[Back to Table of Contents]

Sometimes, `ssh` connections can disconnect due to network or hardware problems. Any processes initialized through that connection will be “hung up” and terminate. Running a command with `nohup` insures that the command will not be hung up if the shell is closed or if the network connection fails.

Run `yes` (continually outputs "y" until it's killed) with `nohup`:

```
[ andrew@pc01 ~ ]$ nohup yes &
[1] 13173
```

`ps` shows a list of the current user's processes (note PID number 13173):

```
[ andrew@pc01 ~ ]$ ps | sed -n '/yes/p'
13173 pts/10   00:00:12 yes
```

...log out and log back into this shell...

The process has disappeared from `ps`!

```
[ andrew@pc01 ~ ]$ ps | sed -n '/yes/p'
```

But it still appears in `top` and `htop` output:

```
[ andrew@pc01 ~ ]$ top -bn 1 | sed -n '/yes/p'
13173 andrew    20   0    4372    704    636 D  25.0  0.0   0:35.99 yes
```

Kill this process with `-9` followed by its process ID (PID) number:

```
[ andrew@pc01 ~ ]$ kill -9 13173
```

It no longer appears in `top`, because it's been killed:

```
[ andrew@pc01 ~ ]$ top -bn 1 | sed -n '/yes/p'
```

cron / crontab / >>

[Back to Table of Contents]

`cron` provides an easy way of automating regular, scheduled tasks.

You can edit your `cron` jobs with `crontab -e` (opens a text editor). Append the line:

```
* * * * * date >> ~/datefile.txt
```

This will run the `date` command every minute, appending (with the `>>` operator) the output to a file:

```
[ andrew@pc02 ~ ]$ head ~/datefile.txt
Sat Jan 12 14:37:01 GMT 2019
Sat Jan 12 14:38:01 GMT 2019
Sat Jan 12 14:39:01 GMT 2019...
```

Just remove that line from the `crontab` file to stop the job from running. `cron` jobs can be set up to run at particular minutes of each hour (0-59), particular hours of each day (0-23), particular days of each month (1-31), particular months of each year (1-12), or particular days of each week (0-6, Sun-Sat). This is what the five stars at the beginning of the command above represent, respectively. Replace them with specific numbers to run them on particular days or at particular times.

If a job is to be run irrespective of, for instance, the day of the week, then the position that represents the day of the week (the 5th position) should contain a star (`*`). This is why the command above runs every minute (the smallest interval available). `cron` jobs can be set up to run only when the system is rebooted, with `@reboot` replacing the stars/numbers. Jobs can also be run a specific number of times per hour or day or at multiple specific times per hour / day / week / month / etc.

Check out this tutorial for more info.

Miscellaneous

`pushd / popd`

[Back to Table of Contents]

Use `pushd` and `popd` to maintain a directory stack, instead of `cd` -ing everywhere.

Start in the `home` directory -- this will be the bottom directory in our “stack”:

```
[ andrew@pc01 ~ ]$ pwd
/home/andrew
```

Move to this directory with a long name, “push” it onto the stack with `pushd` :

```
[ andrew@pc01 ~ ]$ pushd /etc/java/security/security.d/
/etc/java/security/security.d ~
```

Move to a third directory and add it to the stack:

```
[ andrew@pc01 security.d ]$ pushd ~/test/
~/test /etc/java/security/security.d ~
```

When a new directory is added to the stack, it is added to the left-hand side of the list printed by `pushd`. To "pop" the top directory off (return to the most recent directory we added), we can use the `popd` command.

“Pop” off the top directory, move to the next one down the stack with `popd`:

```
[ andrew@pc01 test ]$ popd
/etc/java/security/security.d ~

[ andrew@pc01 security.d ]$ pwd
/etc/java/security/security.d
```

Pop another directory off the stack and we’ve back to where we started:

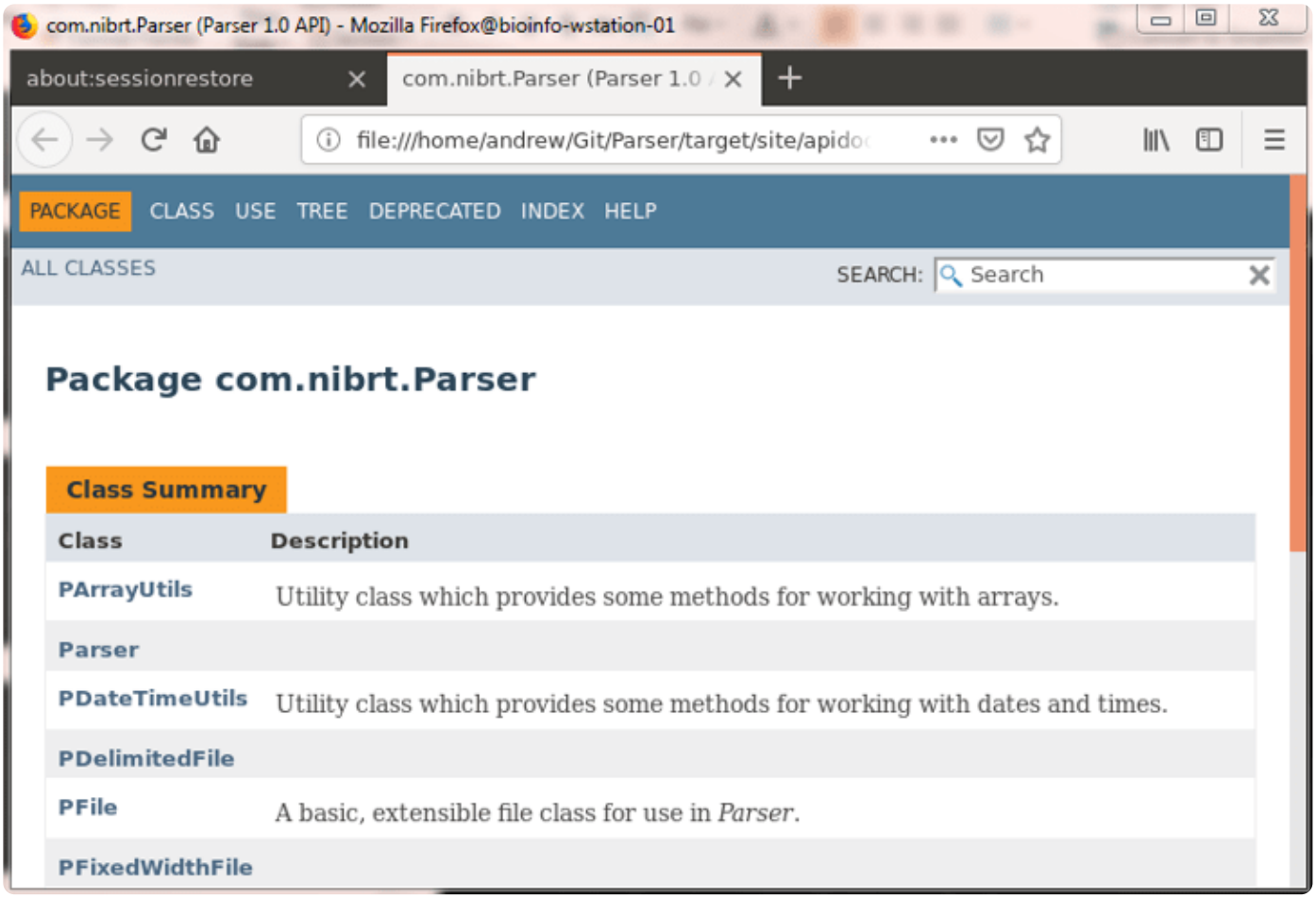
```
[ andrew@pc01 security.d ]$ popd
~

[ andrew@pc01 ~ ]$ pwd
/home/andrew
```

xdg-open

[Back to Table of Contents]

`xdg-open` opens a file with the default application (which could be a GUI program). It's a really useful tool for opening HTML documents from the command line. It's the Unix equivalent of macOS's `open` command:

```
[ andrew@pc01 security.d ]$ xdg-open index.html
```



xargs

[Back to Table of Contents]

`xargs` vectorises commands, running them over any number of arguments in a loop.

`ls` this directory, its parent directory, and its grandparent directory:

```
[ andrew@pc01 ~ ]$ export lv=".\n..\n../.."

[ andrew@pc01 ~ ]$ printf $lv | xargs ls
.:
multi  file

..:
anaid  andrew  colin...

../..:
bin    dev   index...
```

Arguments can be run through a chain of commands with the `-I` flag.

`pwd` this directory, its parent directory, and its grandparent directory by `cd`-ing into each directory first:

```
[ andrew@pc01 ~ ]$ printf $lv | xargs -I % sh -c 'cd %; pwd %'
/home/andrew
/home
/
```

Here's a great tutorial on xargs.

Bonus: Fun But Mostly Useless Things

`w / write / wall / lynx`

[Back to Table of Contents]

`w` is a more detailed `who`, showing who's logged in and what they're doing:

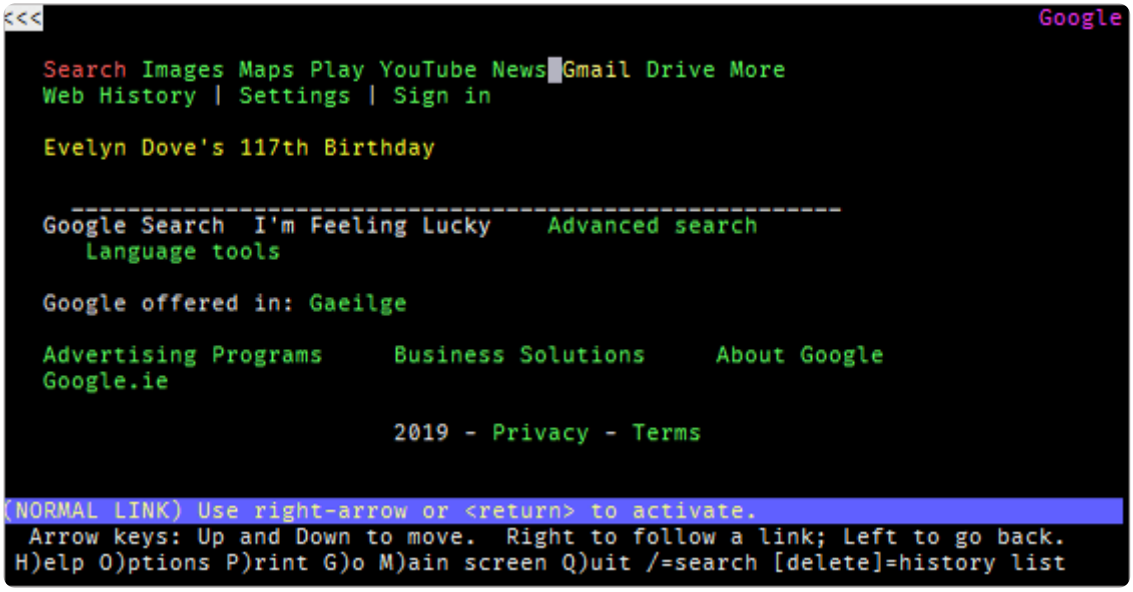
```
[ andrew@pc01 ~ ]$ w
 17:32:42 up 434 days,  3:11,  8 users,  load average: 2.32, 2.46, 2.57
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
colin    pts/9    137.xx.xx.210    03Jan19  5:28m  1:12   0.00s sshd: colin [priv]
andrew   pts/10   137.xx.xx.199    11:05    1.00s  0.15s  0.04s sshd: andrew [priv]
colin    pts/12   137.xx.xx.210    03Jan19 34:32   1.59s  1.59s -bash
...
```

`write` sends a message to a specific user:

```
[ andrew@pc01 ~ ]$ echo "hello" | write andrew pts/10

Message from andrew@pc01 on pts/10 at 17:34 ...
hello
EOF
```

`wall` is similar to `write`, but it sends the same message to every logged-in user. `write` and `wall` used to be more useful before email, Twitter, WhatsApp and instant messaging.

`lynx` is a fully-functional, text-based web browser:



nautilus / date / cal / bc

[Back to Table of Contents]

`nautilus` initialises a GUI remote desktop session and opens a file browser.

`date` shows the current date and time:

```
[ andrew@pc01 ~ ]$ date
Fri Jan 11 17:40:30 GMT 2019
```

`cal` shows an ASCII calendar of this month with today's date highlighted:

```
[ andrew@pc01 ~ ]$ cal
    January 2019
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
```

`bc` is a basic arithmetic calculator (use Python instead):

```
[ andrew@pc01 ~ ]$ bc
bc 1.06.95 ...
20/4
5
```

That's it for now! Let me know if you know of any extra features or cool commands I should add to this list. Also, please let me know if you find any typos or misinformation. I've tried my best to check everything, but there's a lot here!

If you enjoyed this post, please consider supporting my work by buying me a coffee!