Become Efficient at Linux Command Line – Part 1

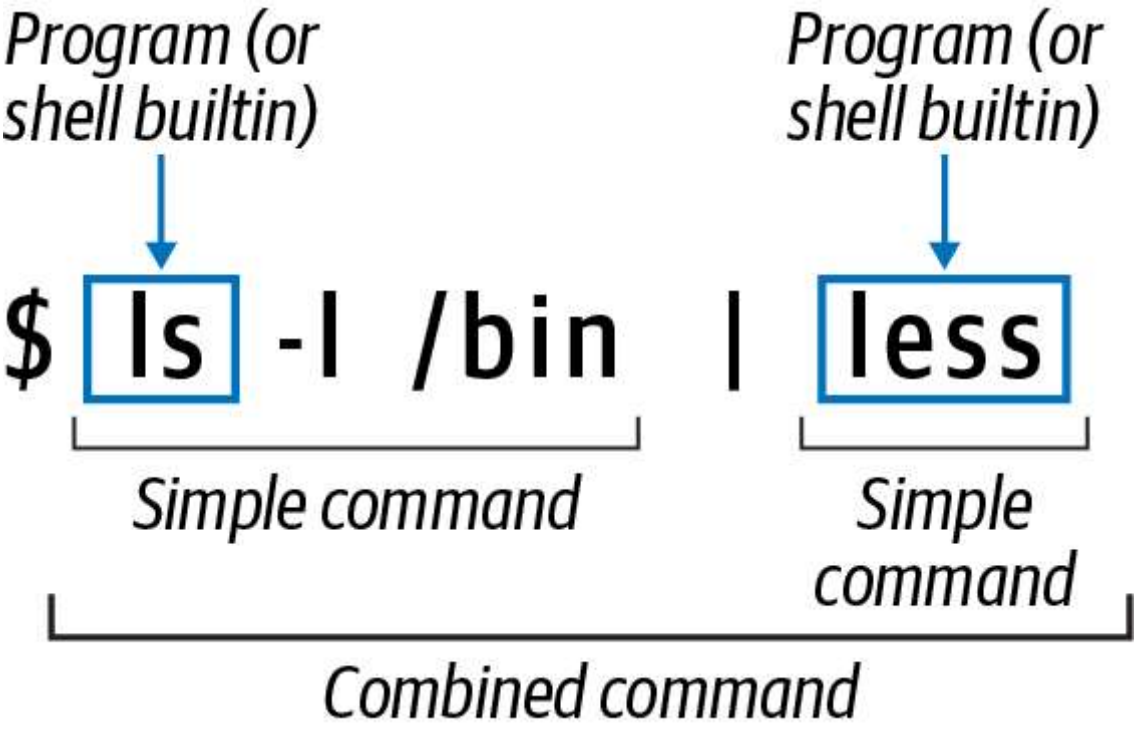
BY KHAJA

Table of Contents

Core Concepts

WHAT'S A COMMAND

The word *command* has three different meanings in Linux, shown below



A program

- An executable program named and executed by a single word, such as ls, or a similar feature built into the shell, such as cd (called a shell builtin)

A simple command

- A program name (or shell builtin) optionally followed by arguments, such as `ls -l /bin`

A combined command

- Several simple commands treated as a unit, such as the pipeline `ls -l /bin | less`

COMBINING COMMANDS

When you work in Windows, macOS, and most other operating systems, you probably spend your time running applications like web browsers, word processors, spreadsheets, and games. A typical

application is packed with features: everything that the designers thought their users would need. So, most applications are self-sufficient. They don't rely on other apps. You might copy and paste between applications from time to time, but for the most part, they're separate

The Linux command line is different. Instead of big applications with tons of features, Linux supplies thousands of small commands with very few features. The command `cat`, for example, prints files on the screen and that's about it. `ls` lists the files in a directory, `mv` renames files, and so on. Each command has a simple, fairly well-defined purpose.

What if you need to do something more complicated? Don't worry. Linux makes it easy to *combine commands* so their individual features work together to accomplish your goal. This way of working yields a very different mindset about computing. Instead of asking "Which app should I launch?" to achieve some result, the question becomes "Which commands should I combine?"

INPUT, OUTPUT, AND PIPES

Most Linux commands read input from the keyboard, write output to the screen, or both. Linux has fancy names for this reading and writing:

stdin (pronounced "standard input" or "standard in")

The stream of input that Linux reads from your keyboard. When you type any command at a prompt, you're supplying data on stdin.

stdout (pronounced "standard output" or "standard out")

The stream of output that Linux writes to your display. When you run the `ls` command to print filenames, the results appear on stdout.

Now comes the cool part. You can connect the stdout of one command to the stdin of another, so the first command feeds the second. Let's begin with the familiar `ls -l` command to list a large directory, such as */bin*, in long format:

```
$ ls -l /bin
total 12104
-rwxr-xr-x 1 root root 1113504 Jun  6  2019 bash
-rwxr-xr-x 1 root root  170456 Sep 21  2019 bsd-csh
-rwxr-xr-x 1 root root   34888 Jul  4  2019 bunzip2
-rwxr-xr-x 1 root root 2062296 Sep 18  2020 busybox
-rwxr-xr-x 1 root root   34888 Jul  4  2019 bzcat
⋮
-rwxr-xr-x 1 root root    5047 Apr 27  2017 znew
```

This directory contains far more files than your display has lines, so the output quickly scrolls off-screen. It's a shame that `ls` can't print the information one screenful at a time, pausing until you press a key to continue. But wait: another Linux command has that feature. The `less` command displays a file one screenful at a time:

```
$ less myfile
```

You can connect these two commands because `ls` writes to stdout and `less` can read from stdin. Use a pipe to send the output of `ls` to the input of `less`:

```
$ ls -l /bin | less
```

This combined command displays the directory's contents one screenful at a time. The vertical bar (`|`) between the commands is the Linux pipe symbol. It connects the first command's stdout to the next command's stdin. Any command line containing pipes is called a *pipeline*.

SIX COMMANDS TO GET YOU STARTED

The six commands—`wc`, `head`, `cut`, `grep`, `sort`, and `uniq`—have numerous options and modes of operation that I'll largely skip for now to focus on pipes. To learn more about any command, run the `man` command to display full documentation. For example:

```
$ man wc
```

To demonstrate our six commands in action, I'll use a file named *animals.txt* that lists some books

```
python     Programming Python    2010  Lutz, Mark
snail SSH, The Secure Shell 2005  Barrett, Daniel
alpaca     Intermediate Perl     2012  Schwartz, Randal
robin MySQL High Availability    2014  Bell, Charles
horse Linux in a Nutshell   2009  Siever, Ellen
donkey     Cisco IOS in a Nutshell    2005  Boney, James
oryx  Writing Word Macros   1999  Roman, Steven
```

Each line contains four facts about book, separated by a single tab character: the animal on the front cover, the book title, the year of publication, and the name of the first author.

Command #1: wc

The `wc` command prints the number of lines, words, and characters in a file:

```
$ wc animals.txt
  7  51 325 animals.txt
```

`wc` reports that the file *animals.txt* has 7 lines, 51 words, and 325 characters. If you count the characters by eye, including spaces and tabs, you'll find only 318 characters, but `wc` also includes the invisible newline character that ends each line.

The options `-l`, `-w`, and `-c` instruct `wc` to print only the number of lines, words, and characters, respectively:

```
$ wc -l animals.txt
7 animals.txt
$ wc -w animals.txt
51 animals.txt
$ wc -c animals.txt
325 animals.txt
```

Counting is such a useful, general-purpose task that the authors of `wc` designed the command to work with pipes. It reads from stdin if you omit the filename, and it writes to stdout. Let's use `ls` to list the contents of the current directory and pipe them to `wc` to count lines. This pipeline answers the question, "How many files are visible in my current directory?"

```
$ ls -1
animals.txt
myfile
myfile2
test.py
$ ls -1 | wc -l
4
```

The option `-1`, which tells `ls` to print its results in a single column, is not strictly necessary here.

wc is the first command you've seen in this chapter, so you're a bit limited in what you can do with pipes. Just for fun, pipe the output of wc to itself, demonstrating that the same command can appear more than once in a pipeline. This combined command reports that the number of words in the output of wc is four: three integers and a filename:

```
$ wc animals.txt
  7  51 325 animals.txt
$ wc animals.txt | wc -w
4
```

Why stop there? Add a third `wc` to the pipeline and count lines, words, and characters in the output "4":

```
$ wc animals.txt | wc -w | wc
      1       1       2
```

The output indicates one line (containing the number 4), one word (the number 4 itself), and two characters. Why two? Because the line "4" ends with an invisible newline character.

ls Changes Its Behavior When Redirected

Unlike virtually every other Linux command, ls is aware of whether stdout is the screen or whether it's been redirected (to a pipe or otherwise). The reason is user-friendliness. When stdout is the screen, ls arranges its output in multiple columns for convenient reading:

```
$ ls /bin
bash         dir          kmod       networkctl        red      tar
bsd-csh      dmesg        less       nisdomainname     rm       tempfile
⋮
```

When stdout is redirected, however, ls produces a single column. I'll demonstrate this by piping the output of ls to a command that simply reproduces its input, such as cat:

```
$ ls /bin | cat
bash
bsd-csh
bunzip2
busybox
⋮
```

This behavior can lead to strange-looking results, as in the following example:

```
$ ls
animals.txt   myfile   myfile2    test.py
$ ls | wc -l
4
```

The first `ls` command prints all filenames on one line, but the second command reports that `ls` produced four lines. If you aren't aware of the quirky behavior of `ls`, you might find this discrepancy confusing.

`ls` has options to override its default behavior. Force `ls` to print a single column with the `-1` option, or force multiple columns with the `-C` option.

Command #2: head

The `head` command prints the first lines of a file. Print the first three lines of *animals.txt* with `head` using the option `-n`:

```
$ head -n3 animals.txt
python     Programming Python    2010  Lutz, Mark
snail SSH, The Secure Shell 2005  Barrett, Daniel
alpaca     Intermediate Perl     2012  Schwartz, Randal
```

If you request more lines than the file contains, head prints the whole file (like cat does). If you omit the -n option, head defaults to 10 lines (-n10).

By itself, head is handy for peeking at the top of a file when you don't care about the rest of the contents. It's a speedy and efficient command, even for very large files, because it needn't read the whole file. In addition, head writes to stdout, making it useful in pipelines. Count the number of words in the first three lines of *animals.txt*:

```
$ head -n3 animals.txt | wc -w
20
```

head can also read from stdin for more pipeline fun. A common use is to reduce the output from another command when you don't care to see all of it, like a long directory listing. For example, list the first five filenames in the */bin* directory:

```
$ ls /bin | head -n5
bash
bsd-csh
bunzip2
busybox
bzcat
```

Command #3: cut

The `cut` command prints one or more columns from a file. For example, print all book titles from *animals.txt*, which appear in the second column:

```
$ cut -f2 animals.txt
Programming Python
SSH, The Secure Shell
Intermediate Perl
MySQL High Availability
Linux in a Nutshell
Cisco IOS in a Nutshell
Writing Word Macros
```

`cut` provides two ways to define what a "column" is. The first is to cut by field (`-f`), when the input consists of strings (fields) each separated by a single tab character. Conveniently, that is exactly the format of the file *animals.txt*. The preceding `cut` command prints the second field of each line, thanks to the option `-f2`.

To shorten the output, pipe it to `head` to print only the first three lines

```
$ cut -f2 animals.txt | head -n3
Programming Python
SSH, The Secure Shell
Intermediate Perl
```

You can also cut multiple fields, either by separating their field numbers with commas:

```
$ cut -f1,3 animals.txt | head -n3
python     2010
snail 2005
alpaca     2012
```

or by numeric range:

```
$ cut -f2-4 animals.txt | head -n3
Programming Python    2010  Lutz, Mark
SSH, The Secure Shell 2005  Barrett, Daniel
Intermediate Perl     2012  Schwartz, Randal
```

The second way to define a "column" for `cut` is by character position, using the `-c` option. Print the first three characters from each line of the file, which you can specify either with commas (`1,2,3`) or as a range (`1-3`):

```
$ cut -c1-3 animals.txt
pyt
sna
alp
rob
hor
don
ory
```

Now that you've seen the basic functionality, try something more practical with `cut` and pipes. Imagine that the *animals.txt* file is thousands of lines long, and you need to extract just the authors' last names. First, isolate the fourth field, author name:

```
$ cut -f4 animals.txt
Lutz, Mark
Barrett, Daniel
Schwartz, Randal
⋮
```

Then pipe the results to `cut` again, using the option `-d` (meaning "delimiter") to change the separator character to a comma instead of a tab, to isolate the authors' last names

```
$ cut -f4 animals.txt | cut -d, -f1
Lutz
Barrett
Schwartz
⋮
```

Command #4: grep

`grep` is an extremely powerful command, but for now I'll hide most of its capabilities and say it prints lines that match a given string. For example, the following command displays lines from *animals.txt* that contain the string `Nutshell`:

```
$ grep Nutshell animals.txt
horse Linux in a Nutshell   2009  Siever, Ellen
donkey     Cisco IOS in a Nutshell    2005  Boney, James
```

You can also print lines that *don't* match a given string, with the `-v` option. Notice the lines containing "Nutshell" are absent:

```
$ grep -v Nutshell animals.txt
python     Programming Python    2010  Lutz, Mark
snail SSH, The Secure Shell 2005  Barrett, Daniel
alpaca     Intermediate Perl     2012  Schwartz, Randal
robin MySQL High Availability    2014  Bell, Charles
oryx  Writing Word Macros   1999  Roman, Steven
```

In general, `grep` is useful for finding text in a collection of files. The following command prints lines that contain the string `Perl` in files with names ending in *.txt*:

```
$ grep Perl *.txt
animals.txt:alpaca      Intermediate Perl       2012    Schwartz,
Randal
essay.txt:really love the Perl programming language, which is
essay.txt:languages such as Perl, Python, PHP, and Ruby
```

In this case, `grep` found three matching lines, one in *animals.txt* and two in *essay.txt*.

`grep` reads stdin and writes stdout, making it great for pipelines. Suppose you want to know how many subdirectories are in the large directory */usr/lib*. There is no single Linux command to provide that answer, so construct a pipeline. Begin with the `ls -l` command:

```
$ ls -l /usr/lib
drwxrwxr-x  12 root root    4096 Mar  1  2020 4kstogram
drwxr-xr-x   3 root root    4096 Nov 30  2020 GraphicsMagick-1.4
drwxr-xr-x   4 root root    4096 Mar 19  2020 NetworkManager
-rw-r--r--   1 root root   35568 Dec  1  2017 attica_kde.so
-rwxr-xr-x   1 root root     684 May  5  2018 cnf-update-db
⋮
```

Notice that `ls -l` marks directories with a `d` at the beginning of the line. Use `cut` to isolate the first column, which may or may not be a `d`:

```
$ ls -l /usr/lib | cut -c1
d
d
d
-
-
⋮
```

Then use `grep` to keep only the lines containing `d`:

```
$ ls -l /usr/lib | cut -c1 | grep d
d
d
d
⋮
```

Finally, count lines with `wc`, and you have your answer, produced by a four-command pipeline—*/usr/lib* contains 145 subdirectories:

```
$ ls -l /usr/lib | cut -c1 | grep d | wc -l
145
```

Command #5: sort

The `sort` command reorders the lines of a file into ascending order (the default):

```
$ sort animals.txt
alpaca     Intermediate Perl     2012  Schwartz, Randal
donkey     Cisco IOS in a Nutshell     2005  Boney, James
horse Linux in a Nutshell   2009  Siever, Ellen
oryx  Writing Word Macros   1999  Roman, Steven
python     Programming Python    2010  Lutz, Mark
robin MySQL High Availability     2014  Bell, Charles
snail SSH, The Secure Shell 2005  Barrett, Daniel
```

or descending order (with the `-r` option):

```
$ sort -r animals.txt
snail SSH, The Secure Shell 2005  Barrett, Daniel
robin MySQL High Availability     2014  Bell, Charles
python     Programming Python    2010  Lutz, Mark
oryx  Writing Word Macros   1999  Roman, Steven
horse Linux in a Nutshell   2009  Siever, Ellen
donkey     Cisco IOS in a Nutshell     2005  Boney, James
alpaca     Intermediate Perl     2012  Schwartz, Randal
```

`sort` can order the lines alphabetically (the default) or numerically (with the `-n` option). I'll demonstrate this with pipelines that cut the third field in *animals.txt*, the year of publication:

```
$ cut -f3 animals.txt                          Unsorted
2010
2005
2012
2014
2009
2005
1999
$ cut -f3 animals.txt | sort -n                Ascending
1999
2005
2005
2009
2010
2012
2014
$ cut -f3 animals.txt | sort -nr               Descending
2014
2012
2010
2009
2005
2005
1999
```

To learn the year of the most recent book in *animals.txt*, pipe the output of `sort` to the input of `head` and print just the first line:

```
$ cut -f3 animals.txt | sort -nr | head -n1
2014
```

Maximum and Minimum Values

`sort` and `head` are powerful partners when working with numeric data, one value per line. You can print the maximum value by piping the data to:

```
... | sort -nr | head -n1
```

and print the minimum value with:

```
... | sort -n | head -n1
```

As another example, let's play with the file */etc/passwd*, which lists the users that can run processes on the system.[4] You'll generate a list of all users in alphabetical order. Peeking at the first five lines, you see something like this:

```
$ head -n5 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
smith:x:1000:1000:Aisha Smith,,,:/home/smith:/bin/bash
jones:x:1001:1001:Bilbo Jones,,,:/home/jones:/bin/bash
```

Each line consists of strings separated by colons, and the first string is the username, so you can isolate the usernames with the `cut` command:

```
$ head -n5 /etc/passwd | cut -d: -f1
root
daemon
bin
smith
jones
```

and sort them:

```
$ head -n5 /etc/passwd | cut -d: -f1 | sort
bin
daemon
jones
root
smith
```

To produce the sorted list of all usernames, not just the first five, replace `head` with `cat`:

```
$ cat /etc/passwd | cut -d: -f1 | sort
```

To detect if a given user has an account on your system, match their username with `grep`. Empty output means no account:

```
$ cut -d: -f1 /etc/passwd | grep -w jones
jones
$ cut -d: -f1 /etc/passwd | grep -w rutabaga          (produces no
output)
```

The `-w` option instructs `grep` to match full words only, not partial words, in case your system also has a username that contains "jones", such as `sallyjones2`.

Command #6: uniq

The `uniq` command detects repeated, adjacent lines in a file. By default, it removes the repeats. I'll demonstrate this with a simple file containing capital letters:

```
$ cat letters
A
A
```

```
A
B
B
A
C
C
C
C
$ uniq letters
A
B
A
C
```

Notice that `uniq` reduced the first three `A` lines to a single `A`, but it left the last `A` in place because it wasn't *adjacent* to the first three.

You can also count occurrences with the `-c` option

```
$ uniq -c letters
      3 A
      2 B
      1 A
      4 C
```

Suppose you have a tab-separated file of students' final grades for a university course, ranging from `A` (best) to `F` (worst):

```
$ cat grades
C     Geraldine
B     Carmine
A     Kayla
A     Sophia
B     Haresh
C     Liam
B     Elijah
B     Emma
A     Olivia
D     Noah
F     Ava
```

You'd like to print the grade with the most occurrences. (If there's a tie, print just one of the winners.) Begin by isolating the grades with `cut` and sorting them:

```
$ cut -f1 grades | sort
A
A
```

```
A
B
B
B
B
C
C
D
F
```

Next, use `uniq` to count adjacent lines:

```
$ cut -f1 grades | sort | uniq -c
      3 A
      4 B
      2 C
      1 D
      1 F
```

Then sort the lines in reverse order, numerically, to move the most frequently occurring grade to the top line:

```
$ cut -f1 grades | sort | uniq -c | sort -nr
      4 B
      3 A
      2 C
      1 F
      1 D
```

and keep just the first line with `head`:

```
$ cut -f1 grades | sort | uniq -c | sort -nr | head -n1
      4 B
```

Finally, since you want just the letter grade, not the count, isolate the grade with `cut`:

```
$ cut -f1 grades | sort | uniq -c | sort -nr | head -n1 | cut -c9
B
```

DETECTING DUPLICATE FILES

Let's combine what you've learned with a larger example. Suppose you're in a directory full of JPEG files and you want to know if any are duplicates:

```
$ ls
image001.jpg  image005.jpg  image009.jpg  image013.jpg  image017.jpg
image002.jpg  image006.jpg  image010.jpg  image014.jpg  image018.jpg
⋮
```

You can answer this question with a pipeline. You'll need another command, `md5sum`, which examines a file's contents and computes a 32-character string called a *checksum*:

```
$ md5sum image001.jpg
146b163929b6533f02e91bdf21cb9563  image001.jpg
```

A given file's checksum, for mathematical reasons, is very, very likely to be unique. If two files have the same checksum, therefore, they are almost certainly duplicates. Here, `md5sum` indicates the first and third files are duplicates:

```
$ md5sum image001.jpg image002.jpg image003.jpg
146b163929b6533f02e91bdf21cb9563  image001.jpg
63da88b3ddde0843c94269638dfa6958  image002.jpg
146b163929b6533f02e91bdf21cb9563  image003.jpg
```

Duplicate checksums are easy to detect by eye when there are only three files, but what if you have three thousand? It's pipes to the rescue. Compute all the checksums, use `cut` to isolate the first 32 characters of each line, and sort the lines to make any duplicates adjacent:

```
$ md5sum *.jpg | cut -c1-32 | sort
1258012d57050ef6005739d0e6f6a257
146b163929b6533f02e91bdf21cb9563
146b163929b6533f02e91bdf21cb9563
17f339ed03733f402f74cf386209aeb3
⋮
```

Now add `uniq` to count repeated lines:

```
$ md5sum *.jpg | cut -c1-32 | sort | uniq -c
      1 1258012d57050ef6005739d0e6f6a257
      2 146b163929b6533f02e91bdf21cb9563
      1 17f339ed03733f402f74cf386209aeb3
      ⋮
```

If there are no duplicates, all of the counts produced by `uniq` will be 1. Sort the results numerically from high to low, and any counts greater than 1 will appear at the top of the output:

```
$ md5sum *.jpg | cut -c1-32 | sort | uniq -c | sort -nr
      3 f6464ed766daca87ba407aede21c8fcc
      2 c7978522c58425f6af3f095ef1de1cd5
      2 146b163929b6533f02e91bdf21cb9563
      1 d8ad913044a51408ec1ed8a204ea9502
      ⋮
```

Now let's remove the nonduplicates. Their checksums are preceded by six spaces, the number one, and a single space. We'll use `grep -v` to remove these lines

```
$ md5sum *.jpg | cut -c1-32 | sort | uniq -c | sort -nr | grep -v "
1 "
      3 f6464ed766daca87ba407aede21c8fcc
      2 c7978522c58425f6af3f095ef1de1cd5
      2 146b163929b6533f02e91bdf21cb9563
```

Finally, you have your list of duplicate checksums, sorted by the number of occurrences, produced by a beautiful six-command pipeline. If it produces no output, there are no duplicate files.

This command would be even more useful if it displayed the filenames of the duplicates, but that operation requires features we haven't discussed yet.

```
$ md5sum *.jpg | grep 146b163929b6533f02e91bdf21cb9563
146b163929b6533f02e91bdf21cb9563  image001.jpg
146b163929b6533f02e91bdf21cb9563  image003.jpg
```

and cleaning up the output with `cut`:

```
$ md5sum *.jpg | grep 146b163929b6533f02e91bdf21cb9563 | cut -c35-
image001.jpg
image003.jpg
```

Introducing the Shell

So, you can run commands at a prompt. But what *is* that prompt? Where does it come from, how are your commands run, and why does it matter?

That little prompt is produced by a program called a *shell*. It's a user interface that sits between you and the Linux operating system. Linux supplies several shells, and the most common (and the standard for this book) is called `bash`.

`bash` and other shells do much more than simply run commands. For example, when a command includes a wildcard (`*`) to refer to multiple files at once:

```
$ ls *.py
data.py    main.py    user_interface.py
```

the wildcard is handled entirely by the shell, not by the program `ls`. The shell evaluates the expression `*.py` and invisibly replaces it with a list of matching filenames *before* `ls` runs. In other words, `ls` *never sees the wildcard*. From the perspective of `ls`, you typed the following command:

```
$ ls data.py main.py user_interface.py
```

It redirects stdin and stdout transparently so the programs involved have no idea they are communicating with each other.

Every time a command runs, some steps are the responsibility of the invoked program, such as `ls`, and some are the responsibility of the shell. Expert users understand which is which. That's one reason they can create long, complex commands off the top of their head and run them successfully. They *already know what the command will do* before they press Enter, in part because they understand the separation between the shell and the programs it invokes.

Rather than cover dozens of shell features, I'll hand you just enough information to carry you to the next step of your learning journey:

- Pattern matching for filenames
- Variables to store values
- Redirection of input and output
- Quoting and escaping to disable certain shell features
- The search path for locating programs to run
- Saving changes to your shell environment

SHELL VOCABULARY

The word *shell* has two meanings. Sometimes it means the *concept* of the Linux shell in general, as in "The shell is a powerful tool" or "`bash` is a shell." Other times it means a specific *instance* of a shell running on a given Linux computer, awaiting your next command.

In this book, the meaning of *shell* should be clear from the context most of the time. When necessary, I'll refer to the second meaning as a *shell instance*, a *running shell*, or your *current shell*.

Some shell instances, but not all, present a prompt so you can interact with them. I'll use the term *interactive shell* to refer to these instances. Other shell instances are noninteractive—they run a sequence of commands and exit.

PATTERN MATCHING FOR FILENAMES

you worked with several commands that accept filenames as arguments, such as `cut`, `sort`, and `grep`. These commands (and many others) accept multiple filenames as arguments. For example, you can search for the word *Linux* in one hundred files at once, named *chapter1* through *chapter100*:

```
$ grep Linux chapter1 chapter2 chapter3 chapter4 chapter5 ...and so
on...
```

Listing multiple files by name is a tedious time-waster, so the shell provides special characters as a shorthand to refer to files or directories with similar names. Many folks call these characters

wildcards, but the more general concept is called *pattern matching* or *globbing*. Pattern matching is one of the two most common techniques for speed that Linux users learn

Most Linux users are familiar with the star or asterisk character (*), which matches any sequence of zero or more characters (except for a leading dot)[1] in file or directory paths:

```
$ grep Linux chapter*
```

Behind the scenes, the shell (not `grep`!) expands the pattern `chapter*` into a list of 100 matching filenames. Then the shell runs `grep`.

Many users have also seen the question mark (`?`) special character, which matches any single character (except a leading dot). For example, you could search for the word *Linux* in chapters 1 through 9 only, by providing a single question mark to make the shell match single digits:

```
$ grep Linux chapter?
```

or in chapters 10 through 99, with two question marks to match two digits:

```
$ grep Linux chapter??
```

Fewer users are familiar with square brackets (`[]`), which request the shell to match a single character from a set. For example, you could search only the first five chapters:

```
$ grep Linux chapter[12345]
```

Equivalently, you could supply a range of characters with a dash:

```
$ grep Linux chapter[1-5]
```

You could also search even-numbered chapters, combining the asterisk and the square brackets to make the shell match filenames ending in an even digit:

```
$ grep Linux chapter*[02468]
```

Any characters, not just digits, may appear within the square brackets for matching. For example, filenames that begin with a capital letter, contain an underscore, and end with an `@` symbol would be matched by the shell in this command:

```
$ ls [A-Z]*_*@
```

Terminology: Evaluating Expressions and Expanding Patterns

Strings that you enter on the command line, such as `chapter*` or `Efficient Linux`, are called *expressions*. An entire command like `ls -l chapter*` is an expression too.

When the shell interprets and handles special characters in an expression, such as asterisks and pipe symbols, we say that the shell *evaluates* the expression.

Pattern matching is one kind of evaluation. When the shell evaluates an expression that contains pattern-matching symbols, such as `chapter*`, and replaces it with filenames that match the pattern, we say that the shell *expands* the pattern.

Patterns are valid almost anywhere that you'd supply file or directory paths on the command line. For example, you can list all files in the directory */etc* with names ending in *.conf* using a pattern:

```
$ ls -1 /etc/*.conf
/etc/adduser.conf
/etc/appstream.conf
⋮
/etc/wodim.conf
```

Be careful using a pattern with a command that accepts just one file or directory argument, such as `cd`. You might not get the behavior you expect:

```
$ ls
Pictures   Poems    Politics
$ cd P*                                  Three directories will match
bash: cd: too many arguments
```

If a pattern doesn't match any files, the shell leaves it unchanged to be passed literally as a command argument. In the following command, the pattern `*.doc` matches nothing in the current directory, so `ls` looks for a filename literally named `*.doc` and fails:

```
$ ls *.doc
/bin/ls: cannot access '*.doc': No such file or directory
```

When working with file patterns, two points are vitally important to remember. The first, as I've already emphasized, is that the shell, not the invoked program, performs the pattern matching. I know I keep repeating this, but I'm frequently surprised by how many Linux users don't know it and develop superstitions about why certain commands succeed or fail.

The second important point is that shell pattern matching applies only to file and directory paths. It doesn't work for usernames, hostnames, and other types of arguments that certain commands accept. You also cannot type (say) `s?rt` at the beginning of the command line and expect the shell to run the `sort` program. (Some Linux commands such as `grep`, `sed`, and `awk` perform their own brands of pattern matching.

Filename Pattern Matching and Your Own Programs

All programs that accept filenames as arguments automatically "work" with pattern matching, because the shell evaluates the patterns before the program runs. This is true even for programs and scripts you write yourself. For example, if you wrote a program `english2swedish` that

translated files from English to Swedish and accepted multiple filenames on the command line, you could instantly run it with pattern matching:

```
$ english2swedish *.txt
```

EVALUATING VARIABLES

A running shell can define variables and store values in them. A shell variable is a lot like a variable in algebra—it has a name and a value. An example is the shell variable `HOME`. Its value is the path to your Linux home directory, such as */home/smith*. Another example is `USER`, whose value is your Linux username, which I'll assume is `smith`.

To print the values of `HOME` and `USER` on stdout, run the command `printenv`:

```
$ printenv HOME
/home/smith
$ printenv USER
smith
```

When the shell evaluates a variable, it replaces the variable name with its value. Simply place a dollar sign in front of the name to evaluate the variable. For example, `$HOME` evaluates to the string `/home/smith`.

The easiest way to watch the shell evaluate a command line is to run the `echo` command, which simply prints its arguments (after the shell is finished evaluating them):

```
$ echo My name is $USER and my files are in $HOME    Evaluating
variables
My name is smith and my files are in /home/smith
$ echo ch*ter9                                       Evaluating a
pattern
chapter9
```

WHERE VARIABLES COME FROM

Variables like `USER` and `HOME` are predefined by the shell. Their values are set automatically when you log in. (More on this process later.) Traditionally, such predefined variables have uppercase names.

You also may define or modify a variable anytime by assigning it a value using this syntax:

```
name=value
```

For example, if you work frequently in the directory */home/smith/Projects*, you could assign its name to a variable:

```
$ work=$HOME/Projects
```

and use it as a handy shortcut with `cd`:

```
$ cd $work
$ pwd
/home/smith/Projects
```

You may supply `$work` to any command that expects a directory:

```
$ cp myfile $work
$ ls $work
myfile
```

When defining a variable, no spaces are permitted around the equals sign. If you forget, the shell will assume (wrongly) that the first word on the command line is a program to run, and the equals sign and value are its arguments, and you'll see an error message:

```
$ work = $HOME/Projects                The shell assumes "work" is a
command
work: command not found
```

A user-defined variable like `work` is just as legitimate and usable as a system-defined variable like `HOME`. The only practical difference is that some Linux programs change their behavior internally based on the values of `HOME`, `USER`, and other system-defined variables. For example, a Linux program with a graphical interface might retrieve your username from the shell and display it. Such programs don't pay attention to an invented variable like `work` because they weren't programmed to do so.

VARIABLES AND SUPERSTITION

When you print the value of a variable with `echo`:

```
$ echo $HOME
/home/smith
```

you might think that the `echo` command examines the `HOME` variable and prints its value. That is *not* the case. `echo` knows nothing about variables. It just prints whatever arguments you hand it. What's really happening is that the shell evaluates `$HOME` before running `echo`. From `echo`'s perspective, you typed

```
$ echo /home/smith
```

This behavior is extremely important to understand, especially as we delve into more complicated commands. The shell evaluates the variables in a command—as well as patterns and other shell constructs—before executing the command.

PATTERNS VERSUS VARIABLES

Let's test your understanding of pattern and variable evaluation. Suppose you're in a directory with two subdirectories, *mammals* and *reptiles*, and oddly, the *mammals* subdirectory contains files named *lizard.txt* and *snake.txt*:

```
$ ls
mammals   reptiles
$ ls mammals
lizard.txt  snake.txt
```

In the real world, lizards and snakes are not mammals, so the two files should be moved to the *reptiles* subdirectory. Here are two proposed ways to do it. One works, and one does not:

```
mv mammals/*.txt reptiles                       Method 1

FILES="lizard.txt snake.txt"
mv mammals/$FILES reptiles                      Method 2
```

Method 1 works because patterns match an entire file path. See how the directory name *mammals* is part of both matches for `mammals/*.txt`:

```
$ echo mammals/*.txt
mammals/lizard.txt mammals/snake.txt
```

So, method 1 operates as if you'd typed the following correct command:

```
$ mv mammals/lizard.txt mammals/snake.txt reptiles
```

Method 2 uses variables, which evaluate to their literal value only. They have no special handling for file paths:

```
$ echo mammals/$FILES
mammals/lizard.txt snake.txt
```

So, method 2 operates as if you'd typed the following problematic command:

```
$ mv mammals/lizard.txt snake.txt reptiles
```

This command looks for the file *snake.txt* in the current directory, not in the *mammals* subdirectory, and fails:

```
$ mv mammals/$FILES reptiles
/bin/mv: cannot stat 'snake.txt': No such file or directory
```

To make a variable work in this situation, use a `for` loop that prepends the directory name *mammals* to each filename:

```
FILES="lizard.txt snake.txt"
for f in $FILES; do
  mv mammals/$f reptiles
done
```

SHORTENING COMMANDS WITH ALIASES

A variable is a name that stands in for a value. The shell also has names that stand in for commands. They're called *aliases*. Define an alias by inventing a name and following it with a equals sign and a command:

```
$ alias g=grep                  A command with no arguments
$ alias ll="ls -l"              A command with arguments: quotes are
required
```

Run an alias by typing its name as a command. When aliases are shorter than the commands they invoke, you save typing time:

```
$ ll                                            Runs "ls -l"
-rw-r--r-- 1 smith smith 325 Jul  3 17:44 animals.txt
$ g Nutshell animals.txt                        Runs "grep Nutshell
animals.txt"
horse   Linux in a Nutshell     2009    Siever, Ellen
donkey  Cisco IOS in a Nutshell 2005    Boney, James
```

You can define an alias that has the same name as an existing command, effectively replacing that command in your shell. This practice is called *shadowing* the command. Suppose you like the `less` command for reading files, but you want it to clear the screen before displaying each page. This feature is enabled with the `-c` option, so define an alias called "less" that runs `less -c`

```
$ alias less="less -c"
```

Aliases take precedence over commands of the same name, so you have now shadowed the `less` command in the current shell.

To list a shell's aliases and their values, run `alias` with no arguments:

```
$ alias
alias g='grep'
alias ll='ls -l'
```

To see the value of a single alias, run `alias` followed by its name:

```
$ alias g
alias g='grep'
```

To delete an alias from a shell, run `unalias`:

```
$ unalias g
```

REDIRECTING INPUT AND OUTPUT

The shell controls the input and output of the commands it runs. You've already seen one example: pipes, which direct the stdout of one command to the stdin of another. The pipe syntax, `|`, is a feature of the shell.

Another shell feature is redirecting stdout to a file. For example, if you use `grep` to print matching lines from the *animals.txt* file

```
$ grep Perl animals.txt
alpaca     Intermediate Perl     2012  Schwartz, Randal
```

You can send that output to a file instead, using a shell feature called *output redirection*. Simply add the symbol > followed by the name of a file to receive the output:

```
$ grep Perl animals.txt > outfile

$ cat outfile
alpaca     Intermediate Perl     2012  Schwartz, Randal
```

You have just redirected stdout to the file *outfile* instead of the display. If the file *outfile* doesn't exist, it's created. If it does exist, redirection overwrites its contents. If you'd rather append to the output file rather than overwrite it, use the symbol >> instead:

```
$ grep Perl animals.txt > outfile                                    $
echo There was just one match >> outfile                           $ cat
outfile
alpaca     Intermediate Perl     2012  Schwartz, Randal
There was just one match
```

Output redirection has a partner, *input redirection*, that redirects stdin to come from a file instead of the keyboard. Use the symbol < followed by a filename to redirect stdin.

Many Linux commands that accept filenames as arguments, and read from those files, also read from stdin when run with no arguments. An example is `wc` for counting lines, words, and characters in a file:

```
$ wc animals.txt
  7  51 325 animals.txt
$ wc < animals.txt
  7  51 325
```

Standard Error (stderr) and Redirection

UIn your day-to-day Linux use, you may notice that some output cannot be redirected by `>`, such as certain error messages. For example, ask `cp` to copy a file that doesn't exist, and it produces this error message:

```
$ cp nonexistent.txt file.txt
cp: cannot stat 'nonexistent.txt': No such file or directory
```

If you redirect the output (stdout) of this `cp` command to a file, *errors*, the message still appears on-screen:

```
$ cp nonexistent.txt file.txt > errors
cp: cannot stat 'nonexistent.txt': No such file or directory
```

and the file *errors* is empty:

```
$ cat errors

```

Why does this happen? Linux commands can produce more than one stream of output. In addition to stdout, there is also stderr (pronounced "standard error" or "standard err"), a second stream of output that is traditionally reserved for error messages. The streams stderr and stdout look identical on the display, but internally they are separate. You can redirect stderr with the symbol `2>` followed by a filename:

```
$ cp nonexistent.txt file.txt 2> errors
$ cat errors
cp: cannot stat 'nonexistent.txt': No such file or directory
```

and append stderr to a file with `2>>` followed by a filename:

```
$ cp nonexistent.txt file.txt 2> errors
$ cp another.txt file.txt 2>> errors
$ cat errors
cp: cannot stat 'nonexistent.txt': No such file or directory
cp: cannot stat 'another.txt': No such file or directory
```

To redirect both stdout and stderr to the same file, use `&>` followed by a filename:

```
$ echo This file exists > goodfile.txt                                  $ cat
goodfile.txt nonexistent.txt &> all.output
$ cat all.output
This file exists
cat: nonexistent.txt: No such file or directory
```

It's *very important* to understand how these two `wc` commands differ in behavior:

- In the first command, `wc` receives the filename *animals.txt* as an argument, so `wc` is aware that the file exists. `wc` deliberately opens the file on disk and reads its contents.
- In the second command, `wc` is invoked with no arguments, so it reads from stdin, which is usually the keyboard. The shell, however, sneakily redirects stdin to come from *animals.txt* instead. `wc` has no idea that the file *animals.txt* exists.

The shell can redirect input and output in the same command:

```
$ wc < animals.txt > count
$ cat count
  7  51 325
```

and can even use pipes at the same time. Here, `grep` reads from redirected stdin and pipes the results to `wc`, which writes to redirected stdout, producing the file *count*:

```
grep Perl < animals.txt | wc > count
$ cat count
      1       6      47
```

DISABLING EVALUATION WITH QUOTES AND ESCAPES

Normally the shell uses whitespace as a separator between words. The following command has four words—a program name followed by three arguments:

```
$ ls file1 file2 file3
```

Sometimes, however, you need the shell to treat whitespace as significant, not as a separator. A common example is whitespace in a filename such as *Efficient Linux Tips.txt*:

```
$ ls -l
-rw-r--r-- 1 smith smith 36 Aug  9 22:12 Efficient Linux Tips.txt
```

If you refer to such a filename on the command line, your command may fail because the shell treats the space characters as separators:

```
$ cat Efficient Linux Tips.txt
cat: Efficient: No such file or directory
cat: Linux: No such file or directory
cat: Tips.txt: No such file or directory
```

To force the shell to treat spaces as part of a filename, you have three options—single quotes, double quotes, and backslashes:

```
$ cat 'Efficient Linux Tips.txt'
$ cat "Efficient Linux Tips.txt"
$ cat Efficient\ Linux\ Tips.txt
```

Single quotes tell the shell to treat every character in a string literally, even if the character ordinarily has special meaning to the shell, such as spaces and dollar signs:

```
$ echo '$HOME'
$HOME
```

Double quotes tell the shell to treat all characters literally except for certain dollar signs and a few others you'll learn later:

```
$ echo "Notice that $HOME is evaluated"                    Double quotes
Notice that /home/smith is evaluated
$ echo 'Notice that $HOME is not'                          Single quotes
Notice that $HOME is not
```

A backslash, also called the *escape character*, tells the shell to treat the next character literally. The following command includes an escaped dollar sign:

```
$ echo \$HOME
$HOME
```

Backslashes act as escape characters even within double quotes:

```
$ echo "The value of \$HOME is $HOME"
The value of $HOME is /home/smith
```

but not within single quotes:

```
$ echo 'The value of \$HOME is $HOME'
The value of \$HOME is $HOME
```

Use the backslash to escape a double quote character within double quotes:

```
$ echo "This message is \"sort of\" interesting"
This message is "sort of" interesting
```

A backslash at the end of a line disables the special nature of the invisible newline character, allowing shell commands to span multiple lines:

```
$ echo "This is a very long message that needs to extend \
onto multiple lines"
This is a very long message that needs to extend onto multiple lines
```

Final backslashes are great for making pipelines more readable

```
$ cut -f1 grades \
  | sort \
  | uniq -c \
  | sort -nr \
```

```
  | head -n1 \
  | cut -c9
```

When used this way, the backslash is sometimes called a *line continuation character*.

A leading backslash before an alias escapes the alias, causing the shell to look for a command of the same name, ignoring any shadowing:

```
$ alias less="less -c" Define an alias
$ less myfile   Run the alias, which invokes less -c
$ \less myfile  Run the standard less command, not the alias
```

LOCATING PROGRAMS TO BE RUN

When the shell first encounters a simple command, such as `ls *.py`, it's just a string of meaningless characters. Quick as a flash, the shell splits the string into two words, "ls" and "*.py". In this case, the first word is the name of a program on disk, and the shell must locate the program to run it.

The program `ls`, it turns out, is an executable file in the directory */bin*. You can verify its location with this command:

```
$ ls -l /bin/ls
-rwxr-xr-x 1 root root 133792 Jan 18  2018 /bin/ls
```

or you can change directories with `cd /bin` and run this lovely, cryptic-looking command:

```
$ ls ls
ls
```

which uses the command `ls` to list the executable file *ls*.

How does the shell locate `ls` in the */bin* directory? Behind the scenes, the shell consults a prearranged list of directories that it holds in memory, called a *search path*. The list is stored as the value of the shell variable `PATH`

```
$ echo $PATH
/home/smith/bin:/usr/local/bin:/usr/bin:/bin:/usr/games:/usr/lib/java/
bin
```

Directories in a search path are separated by colons (`:`). For a clearer view, convert the colons to newline characters by piping the output to the `tr` command, which translates one character into another

```
$ echo $PATH | tr : "\n"
/home/smith/bin
/usr/local/bin
/usr/bin
```

```
/bin
/usr/games
/usr/lib/java/bin
```

The shell consults directories in your search path from first to last when locating a program like `ls`. "Does */home/smith/bin/ls* exist? No. Does */usr/local/bin/ls* exist? Nope. How about */usr/bin/ls*? No again! Maybe */bin/ls*? Yes, there it is! I'll run */bin/ls*." This search happens too quickly to notice.

To locate a program in your search path, use the `which` command:

```
$ which cp
/bin/cp
$ which which
/usr/bin/which
```

or the more powerful (and verbose) `type` command, a shell builtin that also locates aliases, functions, and shell builtins

```
$ type cp
cp is hashed (/bin/cp)
$ type ll
ll is aliased to '/bin/ls -l'
$ type type
type is a shell builtin
```

Your search path may contain the same-named command in different directories, such as */usr/bin/less* and */bin/less*. The shell runs whichever command appears in the earlier directory in the path. By leveraging this behavior, you can override a Linux command by placing a same-named command in an earlier directory in your search path, such as your personal *$HOME/bin* directory.

Search Path and Aliases

When the shell searches for a command by name, it checks if that name is an alias before checking the search path. That's why an alias can shadow (take precedence over) a command of the same name.

The search path is a great example of taking something mysterious about Linux and showing it has an ordinary explanation. The shell doesn't pull commands out of thin air or locate them by magic. It methodically examines directories in a list until it finds the requested executable file.

ENVIRONMENTS AND INITIALIZATION FILES, THE SHORT VERSION

A running shell holds a bunch of important information in variables: the search path, the current directory, your preferred text editor, your customized shell prompt, and more. The variables of a

running shell are collectively called the shell's *environment*. When the shell exits, its environment is destroyed.

It would be extremely tedious to define every shell's environment by hand. The solution is to define the environment once, in shell scripts called *startup files* and *initialization files*, and have every shell execute these scripts on startup. The effect is that certain information appears to be "global" or "known" to all of your running shells.

It's located in your home directory and named *.bashrc* (pronounced "dot bash R C"). Because its name begins with a dot, `ls` doesn't list it by default:

```
$ ls $HOME
apple   banana   carrot
$ ls -a $HOME
.bashrc   apple   banana    carrot
```

If *$HOME/.bashrc* doesn't exist, create it with a text editor. Commands you place in this file will execute automatically when a shell starts up,[5] so it's a great place to define variables for the shell's environment, and other things important to the shell, such as aliases. Here is a sample *.bashrc* file. Lines beginning with # are comments:

```
# Set the search path
PATH=$HOME/bin:/usr/local/bin:/usr/bin:/bin
# Set the shell prompt
PS1='$ '
# Set your preferred text editor
EDITOR=emacs
# Start in my work directory
cd $HOME/Work/Projects
# Define an alias
alias g=grep
# Offer a hearty greeting
echo "Welcome to Linux, friend!"
```

Any changes you make to *$HOME/.bashrc* do not affect any running shells, only future shells. You can force a running shell to reread and execute *$HOME/.bashrc* with either of the following commands:

```
$ source $HOME/.bashrc
$ . $HOME/.bashrc
```

This process is known as *sourcing* the initialization file. If someone tells you to "source your dot-bash-R-C file," they mean run one of the preceding commands.

Rerunning Commands

Suppose you've just executed a lengthy command with a detailed pipeline, like this one from "Detecting Duplicate Files"

```
$ md5sum *.jpg | cut -c1-32 | sort | uniq -c | sort -nr
```

and you want to run it a second time. Don't retype it! Instead, ask the shell to reach back into history and rerun the command. Behind the scenes, the shell keeps a record of the commands you invoke so you can easily recall and rerun them with a few keystrokes. This shell feature is called *command history*. Expert Linux users make heavy use of command history to speed up their work and avoid wasting time.

Similarly, suppose you make a mistake typing the preceding command before you run it, such as misspelling "jpg" as "jg":

```
$ md5sum *.jg | cut -c1-32 | sort | uniq -c | sort -nr
```

To fix the mistake, don't press the Backspace key dozens of times and retype everything. Instead, change the command in place. The shell supports *command-line editing* for fixing typos and performing all sorts of modifications like a text editor can.

This chapter will show you how to save lots of time and typing by leveraging command history and command-line editing.

VIEWING THE COMMAND HISTORY

A *command history* is simply a list of previous commands that you've executed in an interactive shell. To see a shell's history, run the `history` command, which is a shell builtin. The commands appear in chronological order with ID numbers for easy reference. The output looks something like this:

```
$ history
 1000  cd $HOME/Music
 1001  ls
 1002  mv jazz.mp3 jazzy-song.mp3
 1003  play jazzy-song.mp3
 ⋮                                    Omitting 477 lines
 1481  cd
 1482  firefox https://google.com
 1483  history                  Includes the command you just ran
```

The output of `history` can be hundreds of lines long (or more). Limit it to the most recent commands by adding an integer argument, which specifies the number of lines to print:

```
$ history 3            Print the 3 most recent commands
 1482  firefox https://google.com
 1483  history
 1484  history 3
```

Since `history` writes to stdout, you also can process the output with pipes. For example, view your history a screenful at a time:

```
$ history | less              Earliest to latest entry
$ history | sort -nr | less   Latest to earliest entry
```

or print only the historical commands containing the word `cd`:

```
$ history | grep -w cd
 1000  cd $HOME/Music
 1092  cd ..
 1123  cd Finances
 1375  cd Checking
 1481  cd
 1485  history | grep -w cd
```

To clear (delete) the history for the current shell, use the `-c` option:

```
$ history -c
```

RECALLING COMMANDS FROM THE HISTORY

Three time-saving ways to recall commands from a shell's history:

Cursoring:

Extremely simple to learn but often slow in practice

History expansion

Harder to learn (frankly, it's cryptic) but can be very fast

Incremental search

Both simple and fast

Each method is best in particular situations, so I recommend learning all three. The more techniques you know, the better you can choose the right one in any situation.

CURSORING THROUGH HISTORY

To recall your previous command in a given shell, press the up arrow key. It's that simple. Keep pressing the up arrow to recall earlier commands in reverse chronological order. Press the down

arrow to head in the other direction (toward more recent commands). When you reach the desired command, press Enter to run it.

Cursoring through the command history is one of the two most common speedups that Linux users learn. Cursoring is efficient if your desired command is nearby in the history—no more than two or three commands in the past—but it's tedious to reach commands that are further away. Whacking the up arrow 137 times gets old quickly.

The best use case for cursoring is recalling and running the immediately previous command.

Frequently Asked Questions About Command History

How many commands are stored in a shell's history?

> The maximum is five hundred or whatever number is stored in the shell variable `HISTSIZE`, which you can change:
>
> ```
> $ echo $HISTSIZE
> 500
> $ HISTSIZE=10000
> ```
>
> Computer memory is so cheap and plentiful that it makes sense to set `HISTSIZE` to a large number so you can recall and rerun commands from the distant past. (A history of 10,000 commands occupies only about 200K of memory.) Or be daring and store unlimited commands by setting the value to `-1`.

What text is appended to the history?

The shell appends exactly what you type, unevaluated. If you run `ls $HOME`, the history will contain "ls $HOME", not "ls /home/smith".`

Are repeated commands appended to the history?

The answer depends on the value of the variable `HISTCONTROL`. By default, if this variable is unset, then every command is appended. If the value is `ignoredups` (which I recommend), then repeated commands are not appended if they are consecutive

```
$ HISTCONTROL=ignoredups
```

Does each shell have a separate history, or do all shells share a single history?

Each interactive shell has a separate history.

I launched a new interactive shell and it already has a history. Why?

Whenever an interactive shell exits, it writes its history to the file *$HOME/.bash_history* or whatever path is stored in the shell variable `HISTFILE`:

```
$ echo $HISTFILE
/home/smith/.bash_history
```

New interactive shells load this file on startup, so they immediately have a history. It's a quirky system if you're running many shells because they *all* write `$HISTFILE` on exit, so it's a bit unpredictable which history a new shell will load.

The variable `HISTFILESIZE` controls how many lines of history are written to the file. If you change `HISTSIZE` to control the size of the history in memory, consider updating `HISTFILESIZE` as well:

```
$ echo $HISTFILESIZE
500
$ HISTFILESIZE=10000
```

HISTORY EXPANSION

History expansion is a shell feature that accesses the command history using special expressions. The expressions begin with an exclamation point, which traditionally is pronounced "bang." For example, two exclamation points in a row ("bang bang") evaluates to the immediately previous command:

```
$ echo Efficient Linux
Efficient Linux
$ !!                                                    echo
Efficient Linux
```

To refer to the most recent command that began with a certain string, place an exclamation point in front of that string. So, to rerun the most recent `grep` command, run "bang grep":

```
$ !grep
grep Perl animals.txt
alpaca     Intermediate Perl     2012  Schwartz, Randal
```

To refer to the most recent command that contained a given string *somewhere*, not just at the beginning of the command, surround the string with question marks as well

```
$ !?grep?
history | grep -w cd
 1000  cd $HOME/Music
 1092  cd ..
⋮
```

You can also retrieve a particular command from a shell's history by its absolute position—the ID number to its left in the output of `history`. For example, the expression `!1203` ("bang 1023") means "the command at position 1023 in the history":

```
$ history | grep hosts
 1203  cat /etc/hosts
$ !1203                          The command at position 1023
cat /etc/hosts
127.0.0.1       localhost
127.0.1.1       example.directdevops.blog
::1             example.directdevops.blog
```

A negative value retrieves a command by its relative position in the history, rather than absolute position. For example, `!-3` ("bang minus three") means "the command you executed three commands ago":

```
$ history
 4197  cd /tmp/junk
 4198  rm *
 4199  head -n2 /etc/hosts
 4199  cd
 4200  history
$ !-3
head -n2 /etc/hosts
127.0.0.1       localhost
127.0.1.1       example.directdevops.blog
```

History expansion is quick and convenient, if a bit cryptic

INCREMENTAL SEARCH OF COMMAND HISTORY

Wouldn't it be great if you could type a few characters of a command and the rest would appear instantly, ready to run? Well, you can. This speedy feature of the shell, called *incremental search*, is similar to the interactive suggestions provided by web search engines. In most cases, incremental search is the easiest and fastest technique to recall commands from history, even commands you ran long ago. I highly recommend adding it to your toolbox:

1. At the shell prompt, press Ctrl-R (the *R* stands for reverse incremental search).
2. Start typing *any part* of a previous command—beginning, middle, or end.
3. With each character you type, the shell displays the most recent historical command that matches your typing so far.
4. When you see the command you want, press Enter to run it.

Suppose you typed the command `cd $HOME/Finances/Bank` a while ago and you want to rerun it. Press Ctrl-R at the shell prompt. The prompt changes to indicate an incremental search:

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

Start typing the desired command. For example, type `c`:

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

The shell displays its most recent command that contains the string `c`, highlighting what you've typed:

```
(reverse-i-search)`': less /etc/hosts
```

Type the next letter, `d`:

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

The shell displays its most recent command that contains the string `cd`, again highlighting what you've typed:

```
(reverse-i-search)`': cd /usr/local
```

Continue typing the command, adding a space and a dollar sign:

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

The command line becomes:

```
(reverse-i-search)`': cd $HOME/Finances/Bank
```

This is the command you want. Press Enter to run it, and you're done in five quick keystrokes.

I've assumed here that `cd $HOME/Finances/Bank` was the most recent matching command in the history. What if it's not? What if you typed a whole bunch of commands that contain the same string? If so, the preceding incremental search would have displayed a different match, such as:

```
(reverse-i-search)`': cd $HOME/Music
```

What now? You could type more characters to hone in on your desired command, but instead, press Ctrl-R a second time. This keystroke causes the shell to jump to the *next* matching command in the history:

```
(reverse-i-search)`': cd $HOME/Linux/Books
```

Keep pressing Ctrl-R until you reach the desired command:

```
(reverse-i-search)`': cd $HOME/Finances/Bank
```

and press Enter to run it.

Here are a few more tricks with incremental search:

- To recall the most recent string that you searched for and executed, begin by pressing Ctrl-R twice in a row.
- To stop an incremental search and continue working on the current command, press the Escape key, or Ctrl-J, or any key for command-line editing (the next topic in this chapter), such as the left or right arrow key.
- To quit an incremental search and clear the command line, press Ctrl-G or Ctrl-C.

Take the time to become expert with incremental search. You'll soon be locating commands with incredible speed

COMMAND-LINE EDITING

There are all sorts of reasons to edit a command, either while you type it or after you've run it:

- To fix mistakes
- To create a command piece by piece, such as by typing the end of the command first, then moving to the start of the line and typing the beginning
- To construct a new command based on a previous one from your command history 4

Three ways to edit a command to build your skill and speed:

- Cursoring: Again, the slowest and least powerful method but simple to learn
- Caret notation: A form of history expansion
- Emacs- or Vim-style keystrokes: To edit the command line in powerful ways

Cursoring Within a Command

Simply press the left arrow and right arrow keys to move back and forth on the command line, one character at a time. Use the Backspace or Delete key to remove text, and then type any corrections you need. Below Table summarizes these and other standard keystrokes for editing the command line.

Cursoring back and forth is easy but inefficient. It's best when the changes are small and simple.

Keystroke	Action
Left arrow	Move left by one character
Right arrow	Move right by one character
Ctrl + left arrow	Move left by one word
Ctrl + right arrow	Move right by one word

Home	Move to beginning of command line
End	Move to end of command line
Backspace	Delete one character before the cursor
Delete	Delete one character beneath the cursor

History Expansion with Carets

Suppose you've mistakenly run the following command by typing `jg` instead of `jpg`:

```
$ md5sum *.jg | cut -c1-32 | sort | uniq -c | sort -nr
md5sum: '*.jg': No such file or directory
```

To run the command properly, you could recall it from the command history, cursor over to the mistake and fix it, but there's a quicker way to accomplish your goal. Just type the old (wrong) text, the new (corrected) text, and a pair of carets (^), like this:

```
$ ^jg^jpg
```

Press Enter, and the correct command will appear and run:

```
$ ^jg^jpg
md5sum *.jpg | cut -c1-32 | sort | uniq -c | sort -nr
⋮
```

The *caret syntax*, which is a type of history expansion, means, "In the previous command, instead of `jg`, substitute `jpg`." Notice that the shell helpfully prints the new command before executing it, which is standard behavior for history expansion.

This technique changes only the first occurrence of the source string (`jg`) in the command. If your original command contained `jg` more than once, only the first instance would change to `jpg`

Vim-Style Command-Line Editing

The most powerful way to edit a command line is with familiar keystrokes inspired by the text editors Emacs and Vim. If you're already skilled with one of these editors, you can jump into this style of command-line editing right away.

The shell default is Emacs-style editing, and I recommend it as easier to learn and use. If you prefer Vim-style editing, run the following command (or add it to your *$HOME/.bashrc* file and source it):

Action	Emacs	Vim
Move forward by one character	Ctrl-f	h
Move backward by one character	Ctrl-b	l
Move forward by one word	Meta-f	w

Move backward by one word	Meta-b	b
Move to beginning of line	Ctrl-a	0
Move to end of line	Ctrl-e	$
Transpose (swap) two characters	Ctrl-t	xp
Transpose (swap) two words	Meta-t	*n/a*
Capitalize first letter of next word	Meta-c	w~
Uppercase entire next word	Meta-u	*n/a*
Lowercase entire next word	Meta-l	*n/a*
Change case of the current character	*n/a*	~
Insert the next character verbatim, including control characters	Ctrl-v	Ctrl-v
Delete forward by one character	Ctrl-d	x
Delete backward by one character	Backspace *or* Ctrl-h	X
Cut forward by one word	Meta-d	dw
Cut backward by one word	Meta-Backspace *or* Ctrl-w	db
Cut from cursor to beginning of line	Ctrl-u	d^
Cut from cursor to end of line	Ctrl-k	D
Delete the entire line	Ctrl-e Ctrl-u	dd
Paste (yank) the most recently deleted text	Ctrl-y	p
Paste (yank) the next deleted text (after a previous yank)	Meta-y	*n/a*
Undo the previous editing operation	Ctrl-_	u
Undo all edits made so far	Meta-r	U
Switch from insertion mode to command mode	*n/a*	Escape
Switch from command mode to insertion mode	*n/a*	i
Abort an edit operation in progress	Ctrl-g	*n/a*
Clear the display	Ctrl-l	Ctrl-l

Cruising the Filesystem

The techniques in this chapter will help you navigate the filesystem more quickly with less typing. They look deceptively simple but have *enormous* bang for the buck, with small learning curves and big payoffs. These techniques fall into two broad categories:

- Moving quickly to a specific directory
- Returning rapidly to a directory you've visited before

VISITING SPECIFIC DIRECTORIES EFFICIENTLY

If you ask 10 Linux experts what is the most tedious aspect of the command line, seven of them will say, "Typing long directory paths."

After all, if your work files are in */home/smith/Work/Projects/Apps/Neutron-Star/src/include*, your financial documents are in */home/smith/Finances/Bank/Checking/Statements*, and your videos are in */data/Arts/Video/Collection*, it's no fun to retype these paths over and over. In this section, you'll learn techniques to navigate to a given directory efficiently.

Jump to Your Home Directory

Let's begin with the basics. No matter where you go in the filesystem, you can return to your home directory by running `cd` with no arguments:

```
$ pwd
/etc
$ cd
$ pwd
/home/smith
```

To jump to subdirectories within your home directory from anywhere in the filesystem, refer to your home directory with a shorthand rather than an absolute path such as */home/smith*. One shorthand is the shell variable `HOME`:

```
$ cd $HOME/Work
```

Another is a tilde:

```
$ cd ~/Work
```

Both `$HOME` and `~` are expressions expanded by the shell, a fact that you can verify by echoing them to stdout:

```
$ echo $HOME ~
/home/smith /home/smith
```

The tilde can also refer to another user's home directory if you place it immediately in front of their username:

```
$ echo ~jones
/home/jones
```

Move Faster with Tab Completion

When you're entering `cd` commands, save typing by pressing the Tab key to produce directory names automatically. As a demonstration, visit a directory that contains subdirectories, such as */usr*:

```
$ cd /usr
$ ls
bin  games  include  lib  local  sbin  share  src
```

Suppose you want to visit the subdirectory *share*. Type `sha` and press the Tab key once:

```
$ cd sha<Tab>
```

The shell completes the directory name for you:

```
$ cd share/
```

This handy shortcut is called *tab completion*. It works immediately when the text that you've typed matches a single directory name. When the text matches multiple directory names, your shell needs more information to complete the desired name. Suppose you had typed only `s` and pressed Tab:

```
$ cd s<Tab>
```

The shell cannot complete the name *share* (yet) because other directory names begin with `s` too: *sbin* and *src*. Press Tab a second time and the shell prints all possible completions to guide you:

```
$ cd s<Tab><Tab>
sbin/  share/  src/
```

and waits for your next action. To resolve the ambiguity, type another character, `h`, and press Tab once:

```
$ cd sh<Tab>
```

The shell completes the name of the directory for you, from *sh* to *share*:

```
$ cd share/
```

In general, press Tab once to perform as much completion as possible, or press twice to print all possible completions. The more characters you type, the less ambiguity and the better the match.

Hop to Frequently Visited Directories Using Aliases or Variables

If you visit a faraway directory frequently, such as */home/smith/Work/Projects/Web/src/include*, create an alias that performs the `cd` operation:

```
# In a shell configuration file:
alias work="cd $HOME/Work/Projects/Web/src/include"
```

Simply run the alias anytime to reach your destination:

```
$ work
$ pwd
/home/smith/Work/Projects/Web/src/include
```

Alternatively, create a variable to hold the directory path:

```
$ work=$HOME/Work/Projects/Web/src/include
$ cd $work
$ pwd
/home/smith/Work/Projects/Web/src/include
$ ls $work/css
main.css  mobile.css
```

Edit Frequently Edited Files with an Alias

Sometimes, the reason for visiting a directory frequently is to edit a particular file. If that's the case, consider defining an alias to edit that file by absolute path without changing directory. The following alias definition lets you edit *$HOME/.bashrc*, no matter where you are in the filesystem, by running `rcedit`. No `cd` is required

```
# Place in a shell configuration file and source it:
alias rcedit='$EDITOR $HOME/.bashrc'
```

If you regularly visit lots of directories with long paths, you can create aliases or variables for each of them. This approach has some disadvantages, however:

- It's hard to remember all those aliases/variables.
- You might accidentally create an alias with the same name as an existing command, causing a conflict.

An alternative is to create a shell function like the one below

```
# Define the qcd function
qcd () {
  # Accept 1 argument that's a string key, and perform a different
  # "cd" operation for each key.
  case "$1" in
    work)
      cd $HOME/Work/Projects/Web/src/include
      ;;
    recipes)
      cd $HOME/Family/Cooking/Recipes
      ;;
    video)
      cd /data/Arts/Video/Collection
      ;;
    beatles)
      cd $HOME/Music/mp3/Artists/B/Beatles
      ;;
    *)
      # The supplied argument was not one of the supported keys
```

```
      echo "qcd: unknown key '$1'"
      return 1
      ;;
  esac
  # Helpfully print the current directory name to indicate where you
are
  pwd
}
# Set up tab completion
complete -W "work recipes video beatles" qcd
```

Store the function in a shell configuration file such as *$HOME/.bashrc*. source it, and it's ready to run. Type `qcd` followed by one of the supported keys to quickly visit the associated directory:

```
$ qcd beatles
/home/smith/Music/mp3/Artists/B/Beatles
```

As a bonus, the script's final line runs the command `complete`, a shell builtin that sets up customized tab completion for `qcd`, so it completes the four supported keys. Now you don't have to remember `qcd`'s arguments! Just type `qcd` followed by a space and press the Tab key twice, and the shell will print all the keys for your reference, and you can complete any of them in the usual way:

```
$ qcd <Tab><Tab>
beatles  recipes  video    work
$ qcd v<Tab><Enter>
/data/Arts/Video/Collection
```

Make a Big Filesystem Feel Smaller with CDPATH

The `qcd` function handles only the directories that you specify. The shell provides a more general `cd`-ing solution without this shortcoming, called a *cd search path*. This shell feature transformed how I navigate the Linux filesystem.

uppose you have an important subdirectory that you visit often, named *Photos*. It's located at */home/smith/Family/Memories/Photos*. As you cruise around the filesystem, anytime you want to get to the *Photos* directory, you may have to type a long path, such as:

```
$ cd ~/Family/Memories/Photos
```

Wouldn't it be great if you could shorten this path to just *Photos*, no matter where you are in the filesystem, and reach your subdirectory?

```
$ cd Photos
```

Normally, this command would fail:

```
bash: cd: Photos: No such file or directory
```

unless you happen to be in the correct parent directory (~/*Family*/*Memories*) or some other directory with a *Photos* subdirectory by coincidence. Well, with a little setup, you can instruct `cd` to search for your *Photos* subdirectory in locations other than your current directory. The search is lightning fast and looks only in parent directories that you specify. For example, you could instruct `cd` to search *$HOME*/*Family*/*Memories* in addition to the current directory. Then, when you type `cd Photos` from elsewhere in the filesystem, `cd` will succeed:

```
$ pwd
/etc
$ cd Photos
/home/smith/Family/Memories/Photos
```

A cd search path works like your command search path, `$PATH`, but instead of finding commands, it finds subdirectories. Configure it with the shell variable `CDPATH`, which has the same format as `PATH`: a list of directories separated by colons. If your `CDPATH` consists of these four directories, for example:

```
$HOME:$HOME/Projects:$HOME/Family/Memories:/usr/local
```

and you type:

```
$ cd Photos
```

then `cd` will check the existence of the following directories in order, until it finds one or it fails entirely:

- *Photos* in the current directory
- *$HOME*/*Photos*
- *$HOME*/*Projects*/*Photos*
- *$HOME*/*Family*/*Memories*/*Photos*
- */usr*/*local*/*Photos*

In this case, `cd` succeeds on its fourth try and changes directory to *$HOME*/*Family*/*Memories*/*Photos*. If two directories in `$CDPATH` have a subdirectory named *Photos*, the earlier parent wins.

Returning to Directories Efficiently

You've just seen how to visit a directory efficiently. Now I'll show you how to revisit a directory quickly when you need to go back.

Toggle Between Two Directories with "cd -"

Suppose you're working in a deep directory and you run `cd` to go somewhere else:

```
$ pwd
/home/smith/Finances/Bank/Checking/Statements
$ cd /etc
```

and then think, "No, wait, I want to go back to the *Statements* directory where I just was." Don't retype the long directory path. Just run `cd` with a dash as an argument:

```
$ cd -
/home/smith/Finances/Bank/Checking/Statements
```

This command returns your shell to its previous directory and helpfully prints its absolute path so you know where you are.

To jump back and forth between a pair of directories, run `cd -` repeatedly. This is a time-saver when you're doing focused work in two directories in a single shell. There's a catch, however: the shell remembers just one previous directory at a time. For example, if you are toggling between */usr/local/bin* and */etc*:

```
$ pwd
/usr/local/bin
$ cd /etc                  The shell remembers /usr/local/bin
$ cd -                     The shell remembers /etc
/usr/local/bin
$ cd -                     The shell remembers /usr/local/bin
/etc
```