Beyond Code

UNIX, GIT, REGEX, SCRUM, AND MORE

Learn the meta-skills you need to be a better cod no matter *what* lang

FREE SAMPLE

Paul Hudson

Chapter 1

The Unix terminal

Why use the command line?

Every modern operating system has a powerful, user-friendly graphical desktop environment that is capable of doing amazing things. Sure, some people are Linux lovers or Windows fangirls, but ultimately they let you surf the same web pages, edit the same Word documents, and listen to the same music on Spotify.

Why, then, use the command line? It's a text entry system that dates from the 1960s, it has hundreds and even thousands of programs that seem arcane to outsiders, and don't get me started on how badly documented it is. Despite those huge flaws, the command line still delivers incredible power to describe complex operations in a flexible, concise way – you can do things in seconds that would take hours of work in a graphical interface.

I'm going to start by giving you a brief tour of the command line, giving you just enough commands so you can navigate around your system competently. **If you have used the terminal before, you should skip this chapter.** In the subsequent chapters we'll look in more detail at specific commands, and how they can be combined together to create new commands to solve bigger problems.

Note: The command line goes by lots of names: command line, command line interface (CLI), terminal, shell, console, and more. For the sake of brevity, I'll be referring to it as the terminal from now on.

A brief tour of the terminal

Launch your terminal app now, and you'll probably be presented with a black screen, and some white text like this:

```
Last login: Mon May 23 13:39:31 on console
virgil:~ twostraws$
```

The first line just says when we last launched the terminal, but the second line is more important: this is the command prompt. The command prompt appears whenever the system is waiting for you to enter a command, and contains some useful bits of information:

- **`virgil`**: This is the name of my computer – I'm a classics junkie. When you start running commands on other systems, having this reminder is important.
- **`~`**: This tilde is the terminal way of saying "your home directory". More on that in a moment.
- **`twostraws`**: This is my username.
- **`$`**: This divides the prompt from your own typing. Everything after this dollar sign is your text.

Command prompts are often written in different ways, for example you might see this:

```
twostraws@virgil:~$
```

That contains all the same information, just in a different order.

Your home directory is your personal space on your computer, and allows your files to be stored separately from other users'. On Macs, for example, programs live in /Applications, configuration data live in /Library, and user data lives in /Users, for example /Users/twostraws. Inside your home directory will be your desktop (e.g. /Users/twostraws/Desktop), your documents folder (/Users/twostraws/Documents), and so on.

Because the home directory is so commonly used, it gets an alias in the form of **`~`**: rather than writing **`/Users/twostraws/Desktop`** I can just write **`~/Desktop`** to mean the same thing.

Let's run the first command now: type **ls** then hit return. Unix commands are written for brevity, so **ls** is short for **list** – it lists files. By default, **ls** lists the files in the current directory, but you can also list files in other directories. For example, try **ls Desktop** to list the files on your desktop. We're currently in the home directory, so **ls Desktop** will refer to the directory named Desktop inside the home directory.

You can move between directories by using the **cd** command, which is short for "change directory". For example, **cd Desktop** will change to the desktop. So, rather than writing **ls Desktop** to print the contents of the desktop, we could write this:

```
cd Desktop
ls
```

That changes to the Desktop directory, then uses **ls** by itself to print the contents of the current directory. When you change directory, your command prompt will change because you're no longer in **~** – your home directory – and instead you're now in the **Desktop** folder. Your command prompt will now look like one of these:

```
virgil:Desktop twostraws$
twostraws@virgil ~/Desktop$
```

The former shows you only the name of the current directory, whereas the latter shows you the full path: **~/Desktop**. If your command prompt doesn't show the full path, you can use the **pwd** command ("print working directory") to see the path.

You can move from **~/Desktop** back up to **~** (your home directory) by using the **cd** command again, but this time I want to demonstrate a few different options.

1. No matter where you are, you can always move to the parent directory – i.e., the directory one level above where you are right now – by writing **cd ..** – that's two periods.
2. You know that **~** means "home directory", so when you want to go back to the home directory you can just type **cd ~**.
3. **cd** automatically keeps track of your previous directory. If you want to return to it, use **cd -** – that's a dash.

4. Moving to your home directory is in fact so common that just typing **`cd`** by itself with no directory name goes back to your home directory.

The special **`..`** usage is called a pseudo-directory – it looks like a directory name, but it isn't one really, and instead is understood to mean "the parent directory." There's one other pseudo-directory, **`.`**, which means "the current directory." This is important for security reasons: if you downloaded a program called "ls", the system wouldn't run it when you typed "ls" – it would run the system **`ls`** command. If you really wanted to run the one you downloaded, you need to use the **`.`** pseudo-directory, like this: **`./ls`**. Without this precaution, you could accidentally run malware that happened to have the same name as a system command.

You might have noticed that **`.`** and **`..`** don't appear when you run **`ls`**. This is because any files that start with "." are considered to be hidden – you need to instruct **`ls`** to show hidden files in order to show **`.`** and **`..`** and I'll talk more on that later.

Pro tip: If you want to impress your Unix friends, try out the **`pushd`** and **`popd`** commands. Use **`pushd`** like you use **`cd`**, e.g. **`pushd Desktop`**. It changes into that directory, but also remembers all the previous directories you have used. You can then unwind the directory stack by running **`popd`** by itself, which returns you to the previous directory you were in. Use **`popd`** again and again to return to successive previous directories.

Easier navigation

You can now list files and move between directories, so you have enough information to start learning more advanced commands. Before we do that, though, I want to explain a few useful tips that will make your terminal life easier.

First, you need to get friendly with your Tab key. It's usually on the left edge of the keyboard next to Q, and in the terminal it means "complete what I'm typing." So, rather than typing **`cd Desktop`** you can in fact type **`cd De`** then press Tab to have the terminal write the rest of the word for you. If there are multiple possible matches, the terminal will complete as much as it can.

Second, if you want to re-run a command you typed previously, you can use the up and down cursor keys to browse through previous commands. When you find one you want, just hit

return to run it again. If you have run *lots* of commands, you can search instead: press Ctrl+r (hold down Ctrl then press the "r" key), to enter search mode, then start typing to match previous commands. Again, when you find the one you want, just hit return to run it.

Third, although the manual pages for terminal commands are often long and even incoherent, there is one shorter command that is useful: **`whatis`** tells you the purpose of a single command, effectively summarizing the manual page in one line of text. For example, there's a command called "mkdir" that makes directories, but if you weren't sure what it did you could run **`whatis mkdir`** and the system would report "whatis(1) – make directories." (If you were wondering, **`mkdir somedir`** makes a new directory named "somedir".)

Fourth, if you want to execute a long-running command but don't want to wait for it to finish, you can add an ampersand to the end of the command to make it run in the background. The command will appear to finish immediately and you'll be able to type in new commands, but in reality it's still running in the background.

Finally, commands that affect parts of the system outside of your user account require administrator access. For example, if you try to create a directory outside of your home directory, the command will fail because you don't have permission. If you are a system administrator – which, if it's your own computer, you almost certainly are – you can run the command using "super user" privileges, which will ask for your password and run the command as an administrator. To do this, first ensure the command is safe: if you move or delete critical files, your system will break. Once you're sure it's safe, type the same command again, but prefix it with **`sudo`**, which is short for "super user do". For example:

```
sudo ls
```

That will print the contents of the current directory, but will do it as an administrator. The result will be identical, but it's a safe way to test out running commands as an administrator.

That wraps up our gentle tour around the terminal. From here on I want to demonstrate individual commands in more depth so you can learn more about the power of Unix.

Reading file contents

Let's start by looking at some basic commands to work with the contents of files, starting with **cat**, which concatenates files. In practice this means it prints out the contents of one or more files, so to test this out I'd like you to create two files on your desktop, using whatever text editor you like:

1. filea.txt: Give this the text "He thrusts his fists against the posts"
2. fileb.txt: Give this the text "and still insists he sees the ghosts."

Important: after each line of text, I would like you to press return to create a blank second line. Now try running this command:

```
cat filea.txt
```

You should see the "He thrusts his fists" message, because **cat** has loaded the file and printed its contents. If you're curious why the blank line was needed in each file, try removing it from filea.txt then re-running the **cat** command – you'll see this:

```
He thrusts his fists against the postsvirgil:Desktop twostraws$
```

Without the line break, the command prompt starts on the same line as the file's contents, which make for hard reading – so please put the line break back.

We can print two at a time, like this:

```
cat filea.txt fileb.txt
```

The contents of each file just gets sent to output, one after the other. This is helpful because the terminal allows you to redirect your output so that it's written to a file instead of printed for the user to read. To do this, use **>** followed by a filename. For example, this command creates filec.txt by merging the contents of filea.txt and fileb.txt:

```
cat filea.txt fileb.txt > filec.txt
```

If you run that command again and again, it will silently recreate filec.txt by merging filea.txt

and fileb.txt. An alternative is to redirect using **>>** which *appends* rather than *overwrites* the file, like this:

```
cat filea.txt fileb.txt >> filec.txt
```

If you run delete the existing filec.txt then run that command three times, filec.txt will have this content:

```
He thrusts his fists against the posts
and still insists he sees the ghosts.
He thrusts his fists against the posts
and still insists he sees the ghosts.
He thrusts his fists against the posts
and still insists he sees the ghosts.
```

The **cat** command has two options that are frequently useful: **-s** removes blank lines, and **-n** numbers the output. The line numbers are counted individually for each file, so printing two one-line files will give them both the line number 1.

In our case, we have a six-line file, filec.txt, that was created by concatenating filea.txt and fileb.txt, so try running this command:

```
cat -n filec.txt
```

You should see output like the below:

```
1 He thrusts his fists against the posts
2 and still insists he sees the ghosts.
3 He thrusts his fists against the posts
4 and still insists he sees the ghosts.
5 He thrusts his fists against the posts
6 and still insists he sees the ghosts.
```

Warning: Redirecting a file back to itself is a bad idea unless you're very careful. For example, if you ran **cat filea.txt > filea.txt** you would end up with a completely

blank file. This is because your terminal prepares the redirect before anything else happens, which means the first thing it does is clear filea.txt. So, by the time the **`cat filea.txt`** part executes, the file is already empty.

Paging through output

In order to demonstrate what it's like working with more text, I'd like you to go to Wikipedia, select a random article, then paste it into filec.txt using a text editor. Any article will do, as long as it has lots of text.

If you want to the contents of filec.txt, you could use **`cat`** like this:

```
cat filec.txt
```

However, the file ought to be quite long, so you'll see it scroll off your screen quickly. Thanks to modern user interfaces, your terminal probably has a scroll bar so you can move back and forth over the output, but in Ye Olden Days once something had scrolled off the screen it was gone for good.

This is where **`less`** comes in: it reads a file in, but allows you to scroll up and down to read its full content more comfortably. This is helpful even with graphical scrollbars, because it means your fingers can stay on the keyboard rather than moving to and from your mouse or trackpad.

Using **`less`** is just like using **`cat`**:

```
less filec.txt
```

Once it's running, use the up and down cursor keys to scroll around, or press "q" to exit.

Simple, right? Well, it turns out that **`less`** is actually extraordinary powerful: it has options for every letter of the alphabet, which means pressing keys from "a" to "z" all do different things, and in fact many letters even do different things depending on whether they are used in uppercase or lowercase.

Don't worry, I'm not going to explain all 52 options, but I do want to explain a handful of the most interesting ones. They can be split into two categories: parameters you pass to **`less`**

when you run it, and commands you run when **less** is running.

Let's start with parameters you pass before you run **less**, because there are only three worth learning. First, use **-N** (not **-n**!) to enable line numbering inside **less**, for example:

```
less -N filec.txt
```

Now, look at the bottom of your **less** output. Although some systems are smart enough to avoid this, the vast majority of people will see a single colon at the bottom of the screen as they scroll around, which is like the command prompt for **less**. This isn't helpful, but you can pass the **-M** (not **-m**!) parameter to **less** to make that change:

```
less -M filec.txt
```

Now you will also see the filename you're viewing, the range of lines currently visible, the total number of lines in the file, and a percentage of how far you are through it.

You can use the two parameters together either by specifying them individually or together. So, both of these are identical:

```
less -N -M filec.txt
less -NM filec.txt
```

The last pre-run **less** parameter you should know is **+**, which lets you send commands that less should run *after* it starts. To explain how that works, we need to start learning about the post-run commands – i.e. commands that work while **less** is running – so let's start with an easy one: press **/** to start searching for text. So, launch **less**, then try **/the** to look for the first instance of the word "the" in your file. You can repeat the search by pressing "/" then return.

So, **+** lets you specify post-run commands to run on the command prompt. For example, you might want to open a file and go to the first mention of "the", in which case you would run **less** like this:

```
less +/the filec.txt
```

This is useful when you're more experienced, because you can create your own commands by saving ones you like.

You've seen that pressing **/** enters search mode, and pressing **/** then return repeats your previous search. Well, if you want to do a *backwards* search (i.e., upwards from your current position), you need to use **?** instead. On most keyboards that's activated by pressing Shift+/, so you can see the link: **/** triggers search, and Shift+/ triggers reverse search.

The **less** command has a few different ways of jumping to a particular point in a file. For example, you can type a number then "g" to go to a particular line, e.g. "50g" will jump to line 50. You can also use "p" to specify a percent, so "50p" will jump to the half-way point in the file.

If you intend to work with the same file for a little while, a more powerful way of navigating is by using markers. These are invisible bookmarks placed inside a file by **less** so you can jump around faster, but they don't get saved so they won't affect the file. To place a marker, press "m" then one of the 52 letters from A-Z and a-z (it's case sensitive!). To jump back to a marker, type **'** (an apostrophe) then the letter you used to place your marker. For example, **ma** places a marker called "a" at the current position, and **'a** jumps back to that marker.

There are two more useful ways to use **less** before we're done. First you can have it read multiple files at the same time just by listing more on the command prompt. For example:

```
less -M filea.txt fileb.txt filec.txt
```

You'll notice I added the **-M** parameter to have the **less** prompt add extra information. That's not required to open multiple files, but it makes life easier. When you have several files open, press **:n** to go to the next file or **:p** to go to the previous one. You can add more open files by using **:e somefile.txt**, and you'll be pleased to know that tab completion works here too. Once you're finished with a file, you can remove it from the list of open files with **:d**.

You can also search across files – i.e., search for a word in any of the files you have open – by using **/***, for example, **/*the** will search for text in any open file. Annoyingly, repeating a cross-file search uses silly keystrokes: you need to press Escape then either "n" (for searching forwards) or "N" (for backwards).

Finally, one last thing: you can launch a terminal from *inside* **`less`** by typing **`!`** then pressing return. You can go ahead and run as many commands as you want, then press Ctrl+D to exit. If you want to run one specific command, e.g. **`ls`**, use this: **`!ls`**. That will run the command and return immediately. If you want to refer to the file you're currently viewing in **`less`**, use **`%`** – that will automatically be replaced with the filename.

Right, enough about **`less`**. Honestly, you'll probably remember only half the things above, and that's OK – everyone uses it in different ways. However, I hope you can appreciate how even a lowly command like **`less`** is actually packed with functionality – and we've covered less than a tenth of it!

Printing parts of files: head and tail

You've seen how **`cat`** prints a whole file, and **`less`** prints a whole file but gives you the ability to scroll around and search. Now let's look at **`head`** and **`tail`**, which are commands that print only the start or end of a file respectively.

By default, both commands print the first or last ten lines of a file. For example, this will print the first ten lines of our file:

```
head filec.txt
```

If you want to read more or fewer lines, use the **`-n`** parameter followed by a number. For example, this will print the final five lines of our file:

```
tail -n 5 filec.txt
```

Where **`tail`** becomes *really* useful is when you use its **`-f`** parameter, because that enables "follow" mode: the program continues to run until you press Ctrl+D, and any new additions to the file automatically get printed out. If you specify two or more filenames for follow mode, **`tail`** will watch them both and tell you when either of them changes.

Counting lines and words

If you want to count the number of lines, words, and characters a file, you should use the **`wc`**

command, like this:

```
wc filec.txt
```

You'll see output like this:

```
26     182    1001 filec.txt
```

The first column shows the number of lines, the second the number of words, and the third the number of characters. The filename is printed at the end because you can count multiple things – and when you do, you'll automatically get a total line for each of the columns.

If you want to count only lines, only words, or only characters, use either **`-l`**, **`-w`**, or **`-c`**. For example, this counts the words in the file:

```
wc -w filec.txt
```

Listing files intelligently

You've already seen the `ls` command, but in order to be a productive terminal user you need to learn two new things: wildcards for filenames, and `ls` parameters.

Let's start with wildcards, because you might already know these from using them elsewhere. There are two that matter: `*` means "any characters", and `?` means "any single character." You can use this to work on a group of files at the same time.

You've already seen that `ls` lists all the files in the current directory. But if you wanted to show only files that start with "D" (Desktop, Documents, etc), you would use this:

```
ls D*
```

You can place that * anywhere in your filename, and have it filter appropriately. Some more examples:

```
ls *.md
ls *.txt *.xml
ls f*e*
```

The first will list only Markdown files, the second only files that end with .txt or .xml, and the third files that begin with the text "f" then any letters, then "e" then any more letters – i.e., "filea.txt" will match.

If you find you need to specify lots of alternatives regularly, you can also use brace expansion like this:

```
ls *.{txt,xml,md}
```

That gets converted into this:

```
ls *.txt *.xml *.md
```

The other wildcard you can use is `?`, which matches any single character. For example, we have three files called filea.txt, fileb.txt and filec.txt, so we could list them all like this:

```
ls file?.txt
```

That expects exactly one character after “file” and before “.txt”, so “file123.txt” won’t match.

So far, so meh. But these wildcards become much more valuable when you realize they are a feature of the *terminal*, not of **`ls`**. This means you can use them with **`cat`**, **`wc`**, **`head`**, and any other commands. For example, because we have three files called filea.txt, fileb.txt, and filec.txt, these commands do the same thing:

```
cat filea.txt fileb.txt filec.txt
cat file?.txt
cat file*.txt
cat file*
```

Options for listing

The **`ls`** command has lots of options to modify the way file lists are shown, but I want to pick out only a few of the most interesting ones.

You already learned that filenames starting with a period are considered to be hidden files. If you want to *show* those hidden files, use **`ls`** with the **`-a`** parameter, like this:

```
ls -a
```

Two parameters you’ll often see used together are **`-l`** and **`-h`**: the former means “show a long listing” so that you see file size, permissions, ownership, and last modified date; the latter means “show file sizes in a way humans can understand.”

Here’s what the output from **`ls -l`** looks like by itself:

```
-rw-r--r--@  1  twostraws  staff  39  23 May 18:43  filea.txt
-rw-r--r--@  1  twostraws  staff  38  23 May 18:37  fileb.txt
-rw-r--r--   1  twostraws  staff  1001  24 May 22:59  filec.txt
```

That contains seven different columns of information, of which you’re likely to care about

only a few. For your curiosity, here's what they all mean:

- `-rw-r--r--@` are the permissions for this file.
- `1` means the number of hard links pointing to this file.
- `twostraws` is the user owner of the file.
- `staff` is the group owner of the file.
- `39` is the size of the file.
- `23 May 18:43` is the last modified time of the file.
- `filec.txt` is the file name being listed.

A couple of those deserve more detail.

Permissions in Unix are described as three sets of values: what can I do with a file, what can people like me (my group) do with a file, and what can everyone else do with a file? Groups matter a lot when you're working on a big system, e.g. where you might have a "student" group and a "teacher" group, but don't matter at all when you're working on a home computer.

Let's break down one of the permission lines: `-rw-r--r--@`. The first `-` means it's a file rather than a directory (directories have `d` there), `rw-` means I can read and write the file but not execute it, `r--` means people in my user group can read it but not write or execute it, the second `r--` means people outside my user group can read it but not write or execute it, and `@` means it has macOS extended attributes. macOS lets programs save attributes about files separately from the file itself, so a text editor might save the position you were at last time you opened a file, or Finder might attach labels.

The number of hard links to a file is also a curious thing. Unix systems allow multiple filenames to point to the same file on disk. So, /Users/twostraws/hello.txt and /etc/hello.txt might be exactly the same file – one isn't an alias for another, they are both real files independently, so if you delete one the other doesn't break. At the same time, they are pointing to the same file, which means if you edit one, the other changes. For directories, the number of hard links will be 2 plus the number of links inside it.

Now let's look at file sizes. My filec.txt has the size 1001, which means it uses 1001 bytes on disk. That's pretty easy to understand if you're looking at small files, but when you see a size like 24721979 you need to read it carefully to understand that it means a 24MB file. This is

where the **-h** parameter comes in, for example:

```
ls -lh
```

```
-rw-r--r--@  1  twostraws  staff  39B   23 May 18:43  filea.txt
-rw-r--r--@  1  twostraws  staff  38B   23 May 18:37  fileb.txt
-rw-r--r--@  1  twostraws  staff  1.0K  24 May 22:59  filec.txt
```

As you can see, using **-h** tells **ls** to use letters like B (bytes) and K (kilobytes) to describe file sizes, which makes for easier reading. You will also see M for megabytes and G for gigabytes.

When you use **ls** with a directory, it will print the contents of that directory automatically. If you want it to work recursively – print the contents of the directory, print the contents of all subdirectories, print the contents of all sub-subdirectories, and so on – use the **-R** parameter.

The most useful options to **ls** are those that sort the output. By default it's sorted alphabetically by name, but there are two other useful options: **-S** (capital S) sorts files by size with the largest shown first, and **-t** (lowercase T) sorts files by when they were last modified with the newest shown first.

You can reverse the sorting by using the **-r** parameter. For example, this first command sorts output largest first, and the second sorts output smallest first, both with long listing so you can see the sorting has worked:

```
ls -lS
ls -lrS
```

Piping one command into another

To help you understand the basics of Unix, I've made a major simplification so far, and I'm going to remove that simplification now. In doing so, you will immediately be able to use the handful of commands you have learned so far in many more ways, so I hope you'll agree it was worth it!

Previously I said "let's look at **head** and **tail**, which are commands that print only the start or end of a file respectively." And that's true. But what I *didn't* say is that the definition of "file" in Unix is broad, and in fact you'll often hear long-term Unix users say that everything is a file. This means things like network sockets are considered files, devices are considered files, the input and output on your screen is a file, and more. Strictly speaking they are called "file descriptors", but the reality is the same: whenever we have been working with file so far, we could be working with almost anything else.

You have already seen how **>** and **>>** can write to and append to files. There's another way of redirecting output, and it's the pipe symbol, **|**. This might be accessed by pressing Shift+\ on your keyboard. This takes the output from one program and redirects it to another.

Let's using **head** as our example. This lists the first 10 lines of a file by default, but now we know anything is a file – including the output from the terminal. This means we can use **head** to display the first 10 lines of output. For example, try to read this command and figure out what it might do:

```
ls -lS | head
```

Let's break it down:

- **ls**: list files
- **-l**: with long listing format
- **S**: sorted largest first
- **|**: send output to...
- **head**: show first 10 items

Combined, that command will show the 10 largest files in the current directory. And how many files are in the current directory? Try this:

```
ls | wc -l
```

Let's break *that* down:

- **`ls`**: list files
- **`|`**: send output to...
- **`wc`**: count words
- **`-l`**: report only number of lines

Combined, that command will count the number of lines returned by **`ls`**, which is the number of files in the current directory.

You can pipe together as many commands as you need, and even though I've only taught you a few commands you can already use them in interesting ways. For example, try figuring out this:

```
ls -S | head -n 50 | less -N
```

You've seen the first part already: list files sorted largest first, then use **`head`** to use only part of the output. This time I specified **`-n 50`** to get the 50 largest files, but that's quite hard to read in the terminal so I piped it *again*. This time it's to **`less`**, and I used the **`-N`** parameter to number the lines of output.

Now that you know how to pipe one command into another, I'll be using it in future chapters.

Author's note: It's at this point where some people can feel a bit lost with the command line – they forget what **`less -N`** does, or why **`ls -r`** is different from **`ls -R`**. Don't worry: I forget this sort of thing all the time, and so will you. The truth is that you only need to remember the options you use regularly; it's important to know that the more obscure options *exist* but if (when!) you forget them, you can just look them up here or in another reference.

Finding files based on search criteria

Now you've seen how many options **ls** and **less** have, it should come as no surprise to you that the command for finding files is a real Swiss army knife of functionality. Again, I'm going to filter down the many options to the handful you'll find most important.

To find files that match search criteria, you use the **find** command. This takes three parameters in its most common usage:

```
find somewhere -iname somefile.txt
```

The first parameter, **somewhere**, is where **file** should look for matches. You can specify **.** to mean the current directory if you want. Either way, **find** always operates recursively, so it will search the directory you specify, as well as all its children, grandchildren, and so on.

The second parameter, **-iname**, means "look for a file named..." and should be followed by the name to look for. The "i" in "iname" means "case-insensitive", which means it will find somefile.txt, Somefile.txt, SOMEFILE.txt, and so on. If you want case-sensitive searching for some reason you can use **-name** instead.

The third parameter, somefile.txt, is actually attached to the second parameter, because that's the name of the file we want **find** to look for. The **find** command can be used to search for things other than a file's name, so you normally specify things in pairs: "look for a name: somename" or "look for a size: somesize".

Important: You can use wildcards with **find**, but you need to be careful. I already said that wildcards are a feature of the shell, not of the commands you run, which means these two commands do different things:

```
find . -iname *.txt
find . -iname "*.txt"
```

In the first command, wildcard expansion is performed by your terminal. In the second, wildcard expansion is performed by **find**. We had the files filea.txt, fileb.txt, and filec.txt earlier, and when you run the first command they would be matched by the terminal. This means the first command is effectively this:

```
find . -iname filea.txt fileb.txt filec.txt
```

That means “find any files with the name filea.txt, fileb.txt, or filec.txt, in the current directory or any subdirectory.” That *might* be what you want, but chance are you mean “find any file that ends in .txt” instead – in which case the second command is the one you want.

You can search using other criteria, with a popular alternative being size. To do this, specify another pair of parameters: **`-size`** followed by a size option. This can be specified in a variety of ways, so here are some examples:

```
find . -size 10k
find . -size +10k
find . -size -10k
find . -size +1G
```

The first line looks for files that are exactly 10 kilobytes, the second for files that are greater than 10 kilobytes, the third for files smaller than 10 kilobytes, and the fourth for files that are greater than 1 gigabyte. Note that “k” is small, but “G” (and “M” for megabytes) are both capital – don’t worry, **`find`** will tell you if you screw that up.

You can combine criteria together, but make sure you specify them in pairs. For example, this is correct:

```
find . -name "*.zip" -size +1G
```

Whereas this will not work:

```
find . -name -size "*.dmg" +1G
```

You can of course pipe the output from **`find`** into other commands. For example, this will count the number of files over 100MB:

```
find . -size +100M | wc -l
```

As you can see, **`wc -l`** is a real workhorse and you’ll find yourself using it a lot!

Executing output

Where **find** starts to get interesting is its ability to execute a command on any files that match your criteria. This uses pretty unpleasant syntax, but it's not hard to learn:

- **-exec** starts the command to execute on each file.
- **{}** (an open brace then a close brace) is replaced by the name of the matching file.
- **\;** (a backslash them a semi-colon) ends the command to execute.

For example, this command finds all text files in the current directory or any subdirectory, and prints their contents out

```
find . -iname "*.txt" -exec cat {} \;
```

You could use that create a combined file that contains all the other text files combined, like this:

```
find . -iname "*.txt" -exec cat {} \; > output
```

Important: If you use this technique, make sure you don't redirect to a file that matches the search criteria, otherwise you'll get into a recursive loop until you run out of disk space!

As an alternative, you can use **-ok** rather than **-exec** to have **find** ask you whether the command should be executed for each matching file:

```
find . -iname "*.txt" -ok cat {} \; > output
```

Searching for text with grep

Now it's time to meet one of the most powerful commands in the Unix toolkit, but for now I'm only going to use a small part of it. The command is called **grep**, and stands for "Globally search a Regular Expression and Print the results." I'll only be covering part of it here because we haven't covered regular expressions yet, so we'll return to **grep** later on.

You've seen how **find** searches for files based on their metadata, such as their name or size. Well, **grep** is used to search for files that contain *content* you specify, which means it's more CPU-intensive because it might need to scan hundred of gigabytes of content.

Let's start with something simple:

```
grep "posts" *
```

That will look through all files in the current directory, and find those that match the search string "posts". In the output, you'll see one line per match, and each line has both the filename and content of the matching line. This makes **grep** super-fast at finding something important, because you can see not only the name of the matching file, but also where the search text occurred in context.

Some terminals enable colored matching by default, but if yours doesn't then add the **--color** option to have your search text highlighted in the output:

```
grep --color "posts" *
```

By default, **grep** searches only the current directory. If you want it to search recursively through subdirectories, the **-r** parameter. It also searches with case sensitivity by default; if you want it to ignore letter case, use the **-i** parameter. We can combine it all together like this:

```
grep -ri --color "posts" *
```

That will recursively search for the case-insensitive text "posts", then print out results in color.

You can change the match information that **grep** prints by using one of three parameters: **-n** adds the line number where each match was found, **-l** prints only the names of matching files, and **-c** prints the name of each file that was searched along with the number of matches in