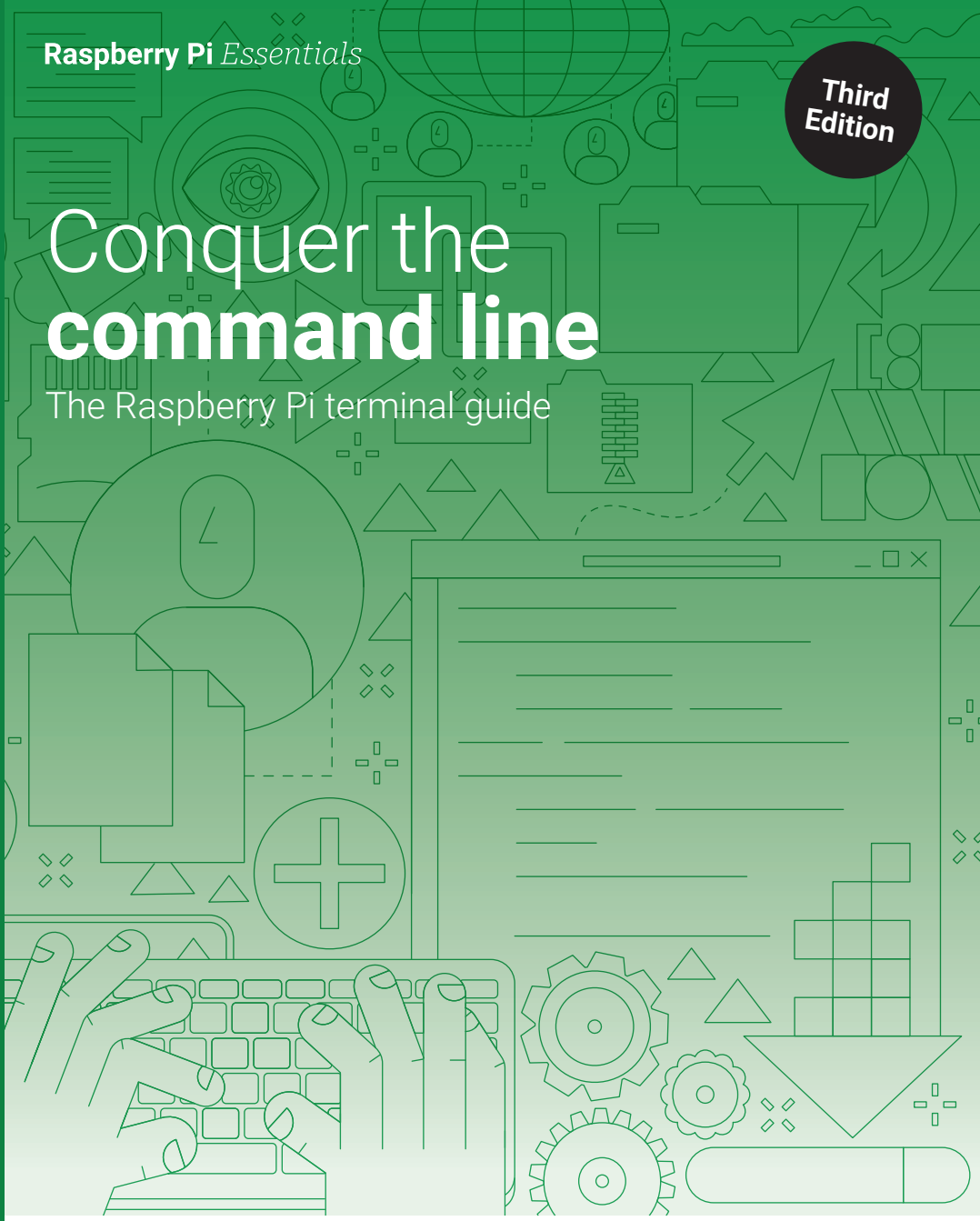
Raspberry Pi *Essentials*

Third Edition

Conquer the **command line**

The Raspberry Pi terminal guide



Raspberry Pi PRESS

Richard Smedley

Conquer the command line, 3rd Edition

Raspberry Pi Essentials: *Conquer the command line*, 3rd Edition
by Richard Smedley
ISBN: 978-1-916868-38-0
Copyright © 2025 Richard Smedley
Printed in the United Kingdom
Published by Raspberry Pi Ltd, 194 Science Park, Cambridge, CB4 0AB
Raspberry Pi Ireland Ltd, 3 Dublin Landings, D01 C4E0, compliance@raspberrypi.com

Editor: Rob Zwetsloot
Copy Editor: Nicola King
Interior Designer: Sara Parodi
Production: Brian Jepson
Photographer: Brian O'Halloran
Illustrator: Sam Alder
Graphics Editor: Natalie Turner
Publishing Director: Brian Jepson
Head of Design: Jack Willis
CEO: Eben Upton

May 2025: Third Edition
February 2019: Second Edition
January 2016: First Edition

The publisher, and contributors accept no responsibility in respect of any omissions or errors relating to goods, products or services referred to or advertised in this book. Except where otherwise noted, the content of this book is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0).

Table of Contents

vii **Welcome**

ix **About the author**

Chapter 1

1 **Don't Panic**

We take a look around and discover that things aren't as strange as they might appear…

Chapter 2

9 **Read and write text**

In this chapter, we get working on files — copying, viewing, removing, editing, and organising

Chapter 3

17 **Permission to install**

We look at the efficient system for installing and updating software in Debian systems

Chapter 4

25 **Manipulating text**

Connect simple commands with pipes for more powerful text processing

Chapter 5

33 **Customise the command line**

Make Raspberry Pi OS a little more personal as you get it to behave and look just the way we want it — for yourself and other users

Chapter 6

39 **Connecting disks**

Disk drives come and disk drives go — tackling the management of removable storage

Chapter 7
47 **Predictable Networking**
In this chapter, we give the Raspberry Pi a permanent network address of its own

Chapter 8
53 **Stopping a process**
As close to perfect as Raspberry Pi OS is, things can go wrong. In this chapter, we learn that there's no need to turn Raspberry Pi off and on again: just kill the process!

Chapter 9
61 **Remote Raspberry Pi**
Learn how to access your Raspberry Pi from remote PCs and devices with Secure Shell (SSH)

Chapter 10
69 **Downloading and installing**
We look at downloading and unpacking software, and show you how to write images to SD cards

Chapter 11
77 **Start and Stop at your Command**
We take a look at scripts to manage the way Raspberry Pi OS starts and shuts down

Chapter 12
87 **Save it now!**
Learn how to protect your data with backups and disk wipes

Chapter 13
99 **Easy compilation**
Here we look at how to install software and build it from the source code

Chapter 14

109 **Commanding the internet**

Yes, it is possible to get online from the command line. Here's how…

Welcome

Sometimes only words will do. Graphical user interfaces (GUIs) were a great advance, creating an easy route into computers for non-technical users. But for complex tasks, GUIs can block off choices and require a circuitous route even for simple tasks.

Enter the command line: the blinking cursor many thought had faded away in the 1990s. For getting instructions from user to computer — in a clear, quick, and unambiguous form — nothing beats the command line. It never disappeared on UNIX and Linux systems, and thanks to Raspberry Pi OS, a new generation is discovering how it simplifies complex tasks and lets you carry out simple ones instantly.

If you're not comfortable when faced with the $ prompt, don't panic! In this fully updated book, we'll quickly make you feel at home and get you familiar with the terminal on Raspberry Pi (or any Mac and Linux computer). Updated for the latest Raspberry Pi software, this book has everything you need to get started.

You can find information about this book, including errata, in its GitHub repository at **rpimag.co/commandlinegit**. If you've found what you believe is a mistake or error in the book, please let us know by using our errata submission form at **rpimag.co/cmdlinefeedback**.

About the author

Since soldering together his first computer — a ZX81 kit — and gaining an amateur radio licence as GW6PCB, Richard has fallen in and out of love with technology. Swapping the ZX81 for a guitar, and dropping ham radio for organic horticulture, he eventually returned to the command line, beginning with a computer to run his own business, and progressing to running all the computers of an international sustainability institution. Now he writes about Free Software and teaches edible landscaping.

Colophon

Raspberry Pi is an affordable way to do something useful, or to do something fun.

Democratising technology — providing access to tools — has been our motivation since the Raspberry Pi project began. By driving down the cost of general-purpose computing to below $5, we've opened up the ability for anybody to use computers in projects that used to require prohibitive amounts of capital. Today, with barriers to entry being removed, we see Raspberry Pi computers being used everywhere, from interactive museum exhibits and schools to national postal sorting offices and government call centres. Kitchen table businesses all over the world have been able to scale and find success in a way that just wasn't possible in a world where integrating technology meant spending large sums on laptops and PCs.

Raspberry Pi removes the high entry cost to computing for people across all demographics: while children can benefit from a computing education that previously wasn't open to them, so too can the many adults who have historically been priced out of using computers for enterprise, entertainment, and creativity. Raspberry Pi eliminates those barriers.

Raspberry Pi Press

store.rpipress.cc

Raspberry Pi Press is your essential bookshelf for all things computing, gaming, and hands-on making. As the official publishing imprint of Raspberry Pi Ltd, we publish a range of titles to help you make the most of your Raspberry Pi hardware. Whether you're building a PC or building a cabinet, you can discover your passion, learn new skills, and make awesome stuff with our extensive collection of books and magazines.

Raspberry Pi Official Magazine

magazine.raspberrypi.com

Raspberry Pi Official Magazine is a monthly publication for makers, engineers, and enthusiasts who love to create with electronics and computer technology. Each issue is packed with Raspberry Pi–themed projects, tutorials, how-to guides, and the latest community news and events.



Chapter 1

Don’t Panic

We take a look around and discover that things aren’t as strange as they might appear…

It’s not a throwback to the past, but a quick and powerful way of getting your Raspberry Pi to do what you want, without all that repetitive strain injury-inducing menu chasing and icon clicking. The command-line interface was a great step up from manually toggling in your instructions in octal (base-8), using switches on the front of the machine! Graphical user interfaces (GUIs) brought friendly visual metaphor to the computer, losing some power and expressiveness. With Raspberry Pi, you can get the best of both worlds by knowing both: after reading through this guide, you’ll soon be as comfortable at the command prompt as you are at your desktop.

By default, Raspberry Pi OS boots you straight to a GUI, although you can change this behaviour in the settings. The command-line environment is still there: hold down the **CTRL+ALT** keys and press **F1** (the first function key on the keyboard), and you’ll arrive at a *virtual console*. Press **CTRL+ALT+F2** through to **F6** and you’ll find five further consoles waiting for you to log in.

You can drop into these any time you like, but for now press **CTRL+ALT+F7** and you’ll be back in mouse and menu land. The command line is also available through a program called a *terminal emulator* (sometimes referred to as a *term*). You’ll also find people referring to the

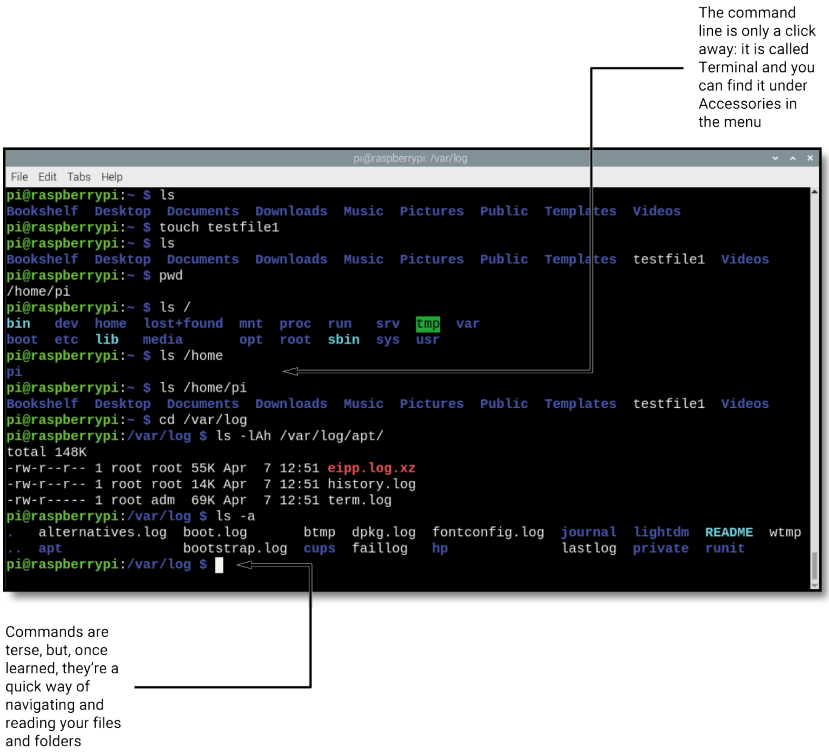


Figure 1-1 Anatomy of your command line

shell, which is the program that runs in the terminal and interprets your commands. On Raspberry Pi and most GNU/Linux systems, the shell is a program named *Bash* (the *Bourne Again SHell*). Don't worry about the distinction between the terminal and shell for now; just click on the icon at the top of the screen that looks like a black television screen, or go to **Accessories > Terminal** in the Raspberry Pi menu: the Terminal now awaits your commands.

Look around

If you're used to looking at files and folders in a file manager, try to clear your mind of the icons and concentrate on the names. Type `ls` and press **RETURN** (see "PRESS RETURN" on page 3). On a fresh Raspberry Pi OS with Recommended Software install, you'll just see a few directories, including **Bookshelf**. Type `ls Bookshelf` and you'll see a listing of what's in it.



TAB COMPLETION

You don't need to type all of `ls Bookshelf` (for example) — after `ls B`, hit the **TAB** key and it will auto-complete. If you've more than one file beginning with **B**, they'll all be listed and you can type more letters and hit **TAB** again.

Commands like `ls` are not cryptic (at least not intentionally) but they are terse, dating back to a time when the connection to the computer was over a 110 baud serial line, from an ASR 33 teletype terminal. If you think it's strange to be defined by 50-year-old technology, just remember that your QWERTY keyboard layout was reputedly designed both to stop mechanical typewriter keys jamming, and to enable salespeople to quickly type 'typewriter' using the top row!



PRESS RETURN

To save repeating it in the text, we'll confirm here that each time you type in a command, you need to hit the **RETURN** or **ENTER** key at the end, to tell the shell that you've issued a command.

File path

You can list files and folders anywhere in your system (or other connected systems) by providing the path as an argument to your command. The path is the folder hierarchy: on a Windows computer, in a graphical file browser, it starts with 'My Computer'; on your Raspberry Pi it starts at `/`, known as *root*, the root of your file system. Try entering `ls /` — again we get terseness, with names like **bin** (short for *binary*) which is the directory where

many programs are kept (try `ls /bin` to see its contents). `ls /dev` shows hardware devices in Raspberry Pi. Try `ls /home` — see your username there? That's you! In this book, we'll show this as 'pi', but you should have chosen your own username when you set up your Raspberry Pi. If you have created extra users, they'll all be listed there too: every user gets their own home directory; ours is the **/home/pi** folder, the working directory where you begin each terminal session. Earlier, with **Bookshelf**, we used the relative path (the absolute path would be **/home/pi/Bookshelf**) because we were already home. If you need to check your current working directory, type `pwd` (*print working directory*).



READ THE MANUAL

Help is included, with man(ual) pages, but they can be a little overwhelming. Use them to check out some extra options beyond the switches like `-a` we use below. To read the *ls* man page, type `man ls`.

There's no place like ~

For any logged-in user, their home directory is abbreviated as `~` (the tilde character). Type `ls ~` and you'll see. There's apparently not much in your home directory yet, but Raspberry Pi OS keeps a lot hidden from the casual glance: files and folders beginning with a dot, known as *dot files*, contain configuration information for your system and its programs. You don't need to see these files normally, but when you do, just ask `ls` to show you all files with a *command switch* (see **Figure 1-2**). You can do this with either the full switch `--all`, or the abbreviation `-a` like so: `ls -a ~`.

Traversing the pathways of the directory hierarchy can be easier from the command line than clicking up and down a directory tree, particularly with all the shortcuts given. Your `ls -a` showed you `.` and `..` as the first two directories; these shortcuts represent the current and the parent directory respectively. Try listing the parent directory from **/home/pi** with `ls ..` — entering `ls ../../` will show you two layers up. If you want to list the hidden files without the `.` and `..` appearing (after all, they're present in every directory, so you don't need to be told), then the switch to use is `-A`.

Before we move on to other commands, try chaining switches together: `ls -lh ~`. Here's what is happening:

`-l`

...gives you more information about the files and folders.

`-h`

...changes the units from bytes to kB, MB, or GB as appropriate.

We'll look at some of the extras the `-l` listing shows you in more detail in Chapter 3, *Permission to install*.

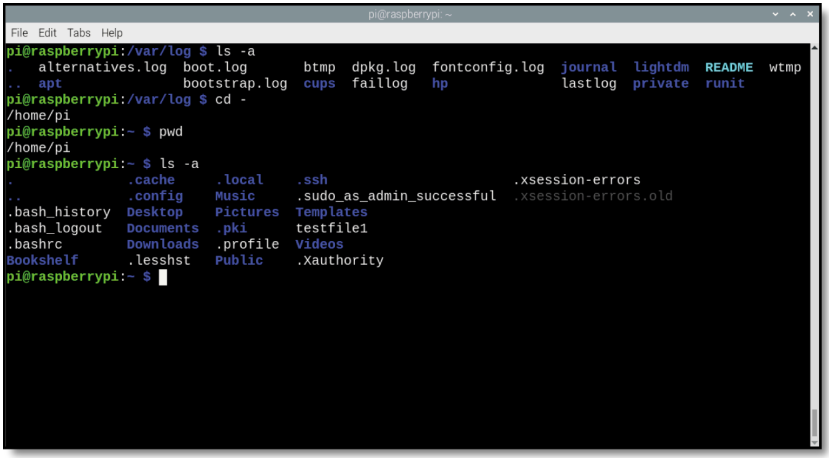


Figure 1-2 `ls -a` shows (dot) files in your listing that are usually hidden from view

Time for change

That's enough looking: let's start moving. `cd` is short for *change directory*, and moves you to anywhere you want in the file system: try `cd /var/log` and have a look (`ls`, remember). Files here are logs, or messages on the state of your system that are saved for analysis later. It's not something you'll often need to think about: Raspberry Pi OS is a version of an operating system (Debian) that also runs across data centres and supercomputers,

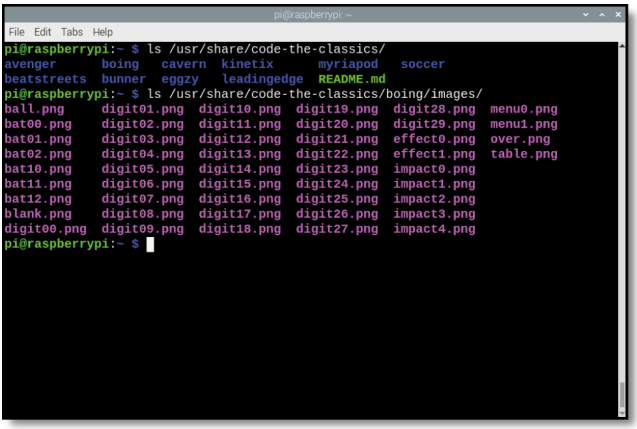
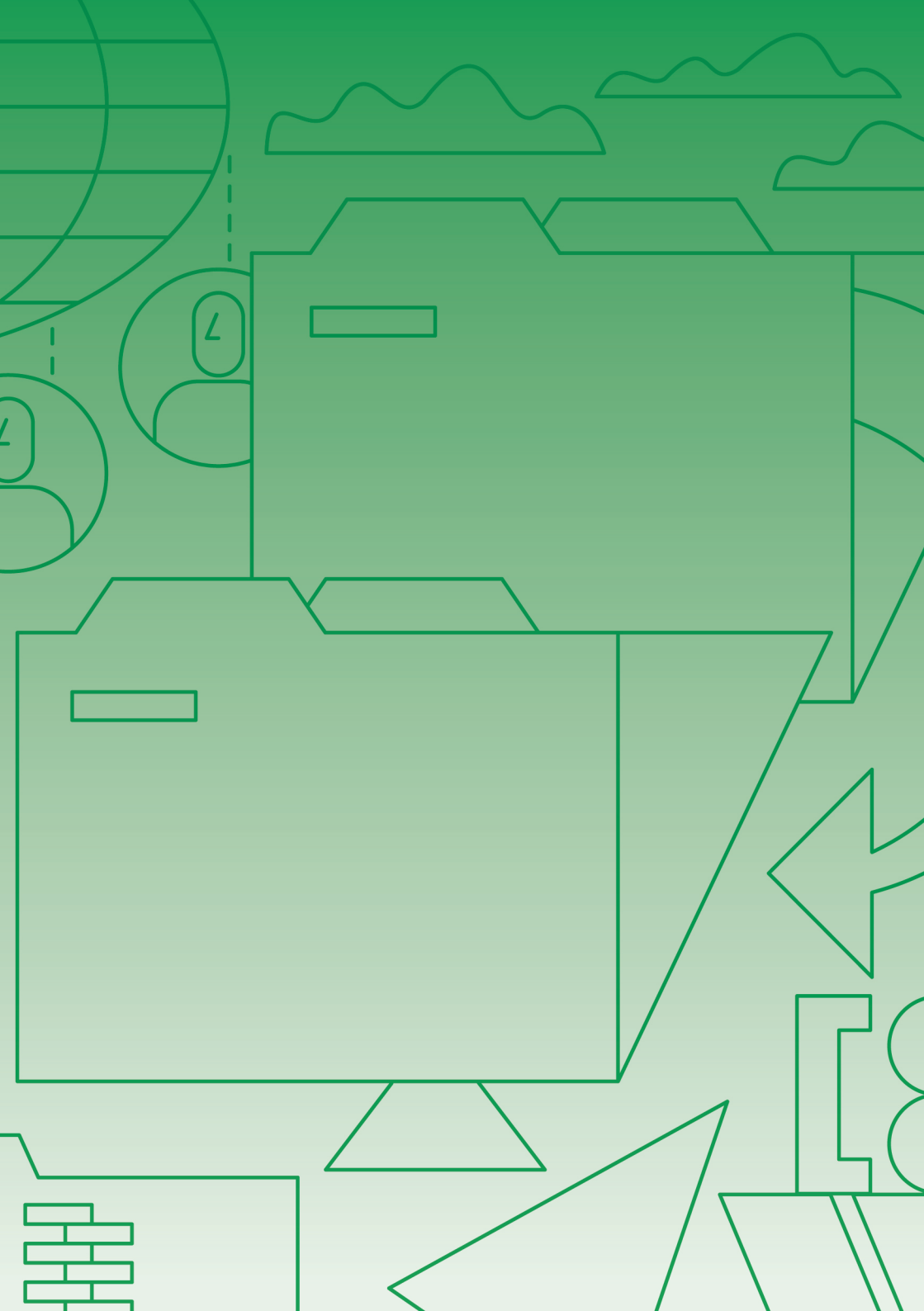


Figure 1-3 Who needs icons when you can fit a listing of files into a small window? Coloured fonts indicate file types

where problem monitoring is very important. It is, however, useful to know, particularly if you have a problem and someone on a forum advises you to check your logs.

`cd ~` will take you where you expect it. Try it, then `pwd` to check. Now try `cd -` (that's a hyphen), the '-' is a shortcut for 'wherever I was before I came here'. Now we've looked around, we can move on to beginning to do things to our files.



Chapter 2

Read and write text

In this chapter, we get working on files — copying, viewing, removing, editing, and organising

Now that we can navigate folders and list files, it's time to learn how to create, view, and alter both files and folders. Once more, it's not that the task is difficult, rather that the forms of the commands (particularly when editing) are unfamiliar, coming from an era before Common User Access (CUA) standards were created to ease switching between applications and operating systems.

Stick with it: with just the first two chapters of this book under your belt, you'll be able to do plenty of work at the command line and start getting comfortable there.

Creating a directory

We're going to dive straight into working with files and folders by creating a new directory. Assuming you already have a terminal open (see "QUICK LAUNCH" on page 11), and you're in your home directory (`pwd` to check, `cd ~` to get back there if necessary), type `mkdir tempfolder` and have a look with `ls`.

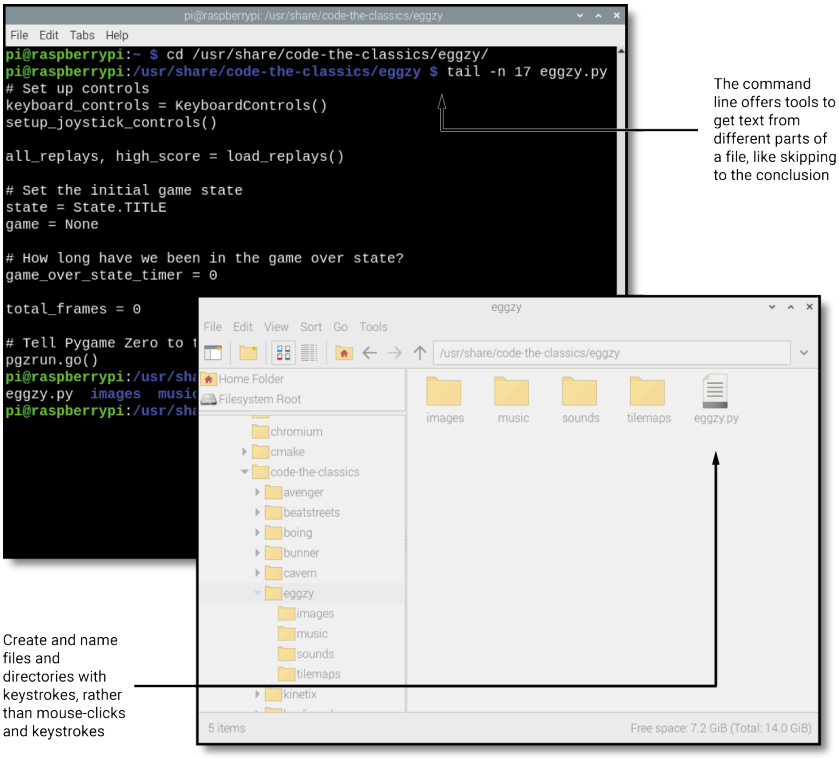


Figure 2-1 Working with the contents of files

`mkdir`, as you've probably guessed, makes a new directory or folder. Let's use it to experiment with altering some Python games. Don't worry: we're not going to be programming Python, just making a small change by way of illustration. `cd tempfolder` (use tab completion: `cd t` then hit the **TAB** key). In the following example, we'll be copying some files to this directory.

First, make sure Code the Classics is installed — if not, click the top-left Raspberry Pi icon on the desktop, select Preferences, then Recommended Software, tick the box next to both Code the Classics I and II in the list, and then click **Apply** to install it.

Next, copy over some files from the **code-the-classics** directory using the `cp` (copy) command:

```
cp /usr/share/code-the-classics/soccer/so* .
cp -r /usr/share/code-the-classics/eggzy .
```

> **QUICK LAUNCH**
>
> Although you can open the terminal emulator from the menu — **Accessories > Terminal** — you can open it instantly by pressing **CTRL+ALT+T**.



Wildcard

The `.` (dot) at the end of the commands refers to your current working directory (**~/tempfolder**), which is where we want the files copied. Also, `so*` is interpreted by the shell as 'every file beginning with `so`' — the `*` is known as a *wildcard*, and this one represents any number of characters (including none); there are other wildcards, including `?`, which means any single character. You'll get an error with the first command:

```
cp: -r not specified; omitting directory
    '/usr/share/code-the-classics/soccer/sounds'
```

This is because `so*` matches both *soccer.py* and *sounds*, but only the first of these is a file; you must use the `-r` (recursive copy) switch to copy directories. The second command uses that switch to make a copy of the entire **eggzy** directory. Remember to use `-r` when copying directories.

Use `cd eggzy` and then try `python eggzy.py` and you'll see you can run the local copy of the game. To change the game, we need an editor — sidestepping the UNIX debate about which one is best, we'll use Raspberry Pi's built-in editor: nano. Unless you've previously used the old Pico editor, which accompanied the Pine email client on many university terminals in the 1980s and 1990s, it will seem a little odd (**Figure 2-2**). That's because its conventions predate standard keyboard shortcuts such as **CTRL+C** and **CTRL+V** found in most modern programs. Nano has unusual com-

mand shortcuts, but they're worth learning, as you'll find nano installed on virtually all GNU/Linux boxes, such as a web host. Bear with us.

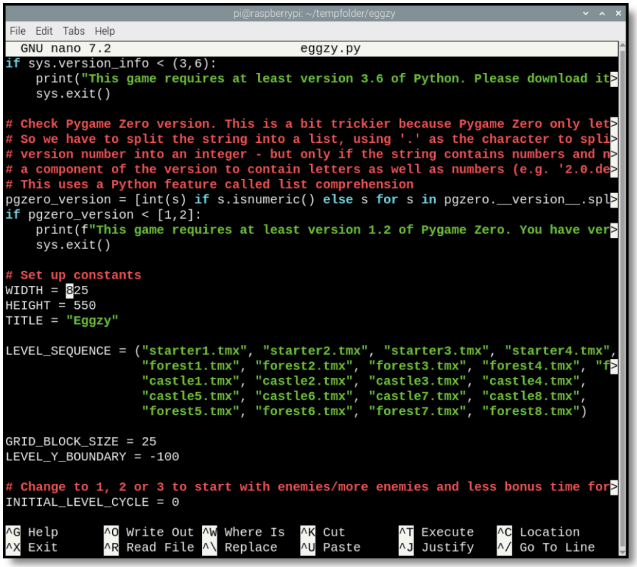


Figure 2-2 The default editor, nano

Editing and paging

`nano eggzy.py` will open the game for editing; use the arrow keys to go down several lines and find the `WIDTH` value of `825`. Change it to `1000` (both the **Backspace** and **Delete** keys will work in nano). The last two lines of the screen show some shortcuts, with `^` (the caret symbol) representing the **CTRL** key: **CTRL+O**, followed by **ENTER** or **RETURN** will 'write out' (save) the file; then use **CTRL+X** to exit. Now, `python eggzy.py` will open an oversize window. However, the main game will not take up the rest of the screen so it's not very useful: go back and reset `WIDTH` to `825`.

If you want to take a look through the *eggzy.py* listing without entering the strange environment of nano, you can see the entire text of any file using `cat`: e.g., `cat eggzy.py`. Unfortunately, a big file quickly scrolls off the screen; to look through a page at a time, you need a 'pager' program.

`less eggzy.py` will let you scroll up and down through the text with the **Page Up** and **Page Down** keys. Other keys will do the same job, but we'll leave you to discover these yourself. To exit `less`, hit **Q** (this also works from man pages, which use a pager to display text).

Cats, heads & tails

If editor wars are a UNIX tradition we can safely ignore, there's no getting away from another tradition: bad puns. `less` is an improvement over `more`, a simple pager; the respective man pages will show you the differences. One advantage the relatively primitive `more` has is that at the end of a file it exits automatically, saving you reaching for the **Q** button. Admittedly, this is not a huge advantage, and you can always use `cat`.

Fortunately, `cat` is not a feline-based pun, but simply short for 'concatenate': use it with more than one file and it concatenates them together. Used with no argument — type `cat` — it echoes back what you type each time you press **ENTER**. Hit **CTRL+C** to exit when you've finished typing in silly words to try it. And remember that **CTRL+C** shortcut: it closes most command-line programs, in the same way that **ALT+F4** closes most windowed programs.

You can peek at the first or last few lines of a text file with `head` and `tail` commands. `head eggzy.py` will show you the first ten lines of the file. `head -n 5 eggzy.py` shows just five lines, as does `tail -n 5 eggzy.py` with the last five lines. With `head`, you can omit the `-n`, so `head -5 eggzy.py` will also work.

SWITCHING HELP



You don't need to wade through the man page to see what switches are available: type `--help` after the command to be shown your options, e.g. `rm --help`.

Remove with care

We've done a lot already (at least, nano makes it feel like a lot), but it's never too early to learn how to clean up after ourselves. We'll remove the files we've created with `rm`. The remove tool should always be used with care: it has some built-in safeguards, but even these are easy to override (**Figure 2-3**). In particular, never let anyone persuade you to type `rm -rf /` — this will delete the entire contents of your Raspberry Pi, all the programs, everything, with little to no chance of recovery.

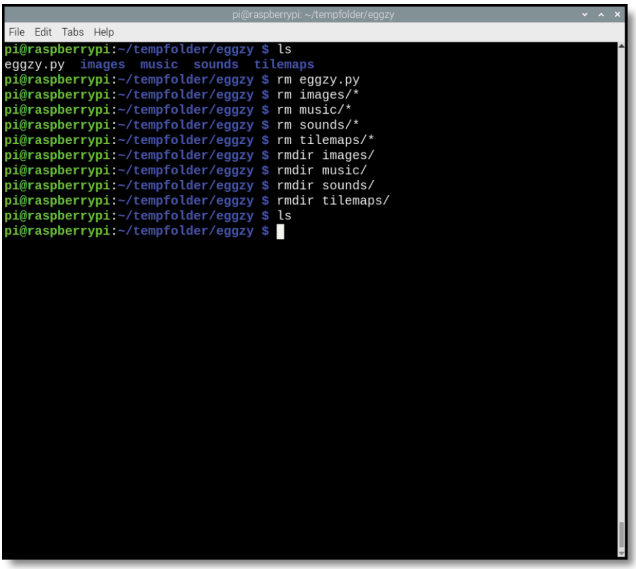


Figure 2-3 `rm` is a powerful removal tool: use with great care!

Have a look at what files we have: if you're still in the **eggzy** subdirectory of the **tempfolder** you made, then `ls` will show you the files you copied here. Remove the program, then the contents of the subdirectories with careful use of the `*` wildcard.

```
rm eggzy.py
rm images/*
rm music/*
rm sounds/*
```

```
rm tilemaps/*
rm eggzy-replays
```

(The **eggzy-replays** file will only exist if you played the game.) Next, remove each of the subdirectories with rmdir:

```
rmdir images
rmdir music
rmdir sounds
rmdir tilemaps
```

If you run the `ls` command, you'll see that the **eggzy** directory is now empty.

`cd ..` to get back to **/home/pi/tempfolder** and `rm -r eggzy` will remove the now empty folder. The `-r` (recursive) option is necessary for directories, and will also remove the contents if any remain. That still leaves the file **soccer.py**. You can delete **tempfolder**, including **soccer.py**, by using `cd ..` to get back to **/home/pi** and running `rm -r tempfolder`. Have a look at "Aliases" on page 37 for some tips on making the `rm`, `cp`, and `mv` (which you can use to move or rename files) commands safer.

In the next chapter, we'll delve into file permissions and updating software from the command line.



Chapter 3

Permission to install

We look at the efficient system for installing and updating software in Debian systems

Installing software should be easy, but behind every piece of software is an evolving set of dependencies that also need installing and updating. Keeping software separate from its dependencies reduces unnecessary bloat and duplication, but adds the potential for bugs, missing files, and even totally unresolvable clashes.

Fortunately, Debian GNU/Linux cracked the problem back in the 1990s with the Debian Package Management system and the Advanced Package Tool (APT) — and Debian-based systems, like Ubuntu and Raspberry Pi OS, inherit all of the benefits. Here we'll show you the basics you need to know to install new software and keep your system up to date from the command line, and then look at the not entirely unrelated field of file ownership and permissions.

Using the `apt` command to update your system's list of installable software should be as simple as issuing the command like so: `apt update`. Try this logged in as user pi, though, and you'll just get error messages. The reason for this is that changing system software on a Unix-like system is a task restricted to those with *administrative permissions*: the godlike superuser, or admin, also known as root.

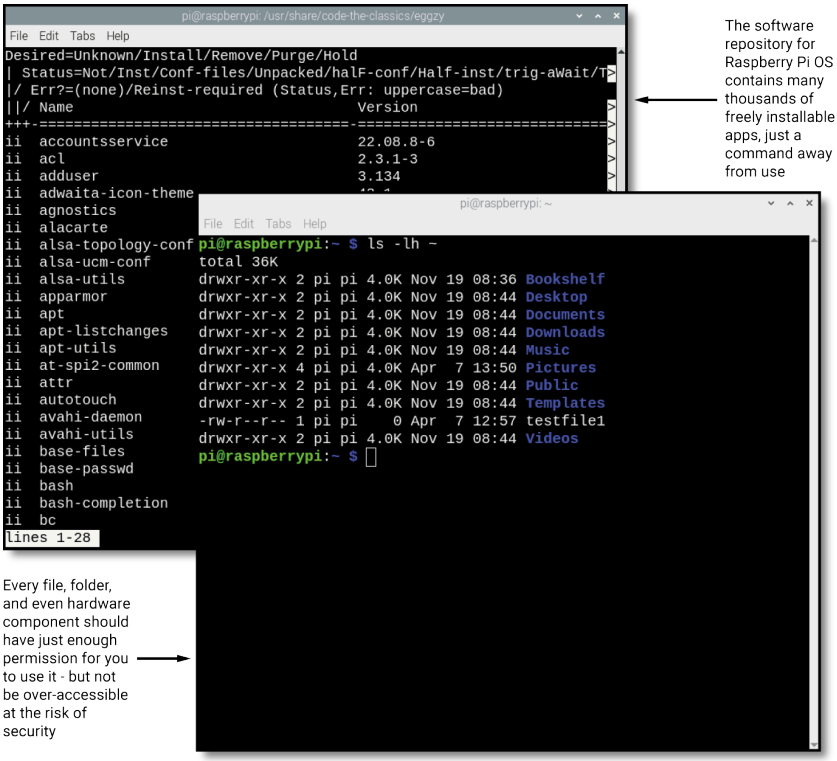


Figure 3-1 The command line gives you power without peril



SHARED RESPONSIBILITY

If you share your Raspberry Pi, read up on `sudo` and the `visudo` command to find how to give limited but useful admin privileges to other users.

Pseudo root, su do

We'll get onto permissions properly a bit later, but for now you'll be pleased to know that you can fake it, using the `sudo` command. `sudo` po-

tentially offers a fine-grained choice of permissions for users and groups to access portions of the admin user's powers. However, on Raspberry Pi OS it assumes, quite rightly, that the default user will be someone just wanting to get on with things, and **sudo** in front of a command will pretty much let you do anything (without prompting you for a password, which would be the norm on any other Unix-like system). You have been warned!

The following commands will update the installed software on Raspberry Pi OS or any apt-based system like Debian or Ubuntu (**Figure 3-2**):

```
sudo apt update
sudo apt upgrade
```

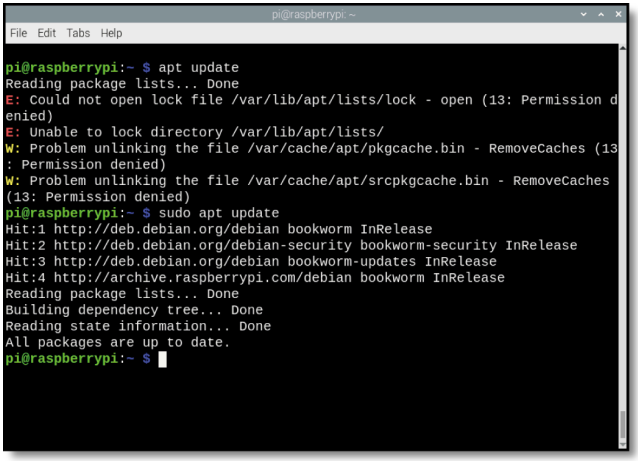


Figure 3-2 Raspberry Pi OS updates its listing of thousands of available apps, providing you give it admin permissions

You can wait for one to finish, check everything is OK, then issue the other command, or you can save waiting and enter both together with:

```
sudo apt update && sudo apt upgrade
```

The **&&** is a Boolean (logical) AND, so if the first command doesn't run properly, the second one will not run at all. This is because for a logical AND to be true, both of its conditions must be true.



SUDO WITH CAUTION

Fine-grained permissions make for greater security, but can trip you up. If you run a command and it gives you an error, you may be tempted to re-run it with `sudo` in the hope that your superuser powers will solve it. But before you do that, take a breath, research the error message you received, and proceed with caution. These restrictive permissions are there for a reason!

It's always worth running the `update` command before installing new software, too — minor updates are made even in stable distributions such as Raspberry Pi, to address any issues. We've just run an update, so no need to repeat that for now. Sticking with a command-line theme, we're going to install an old suite of terminal games:

```
sudo apt install bsdgames
```

When prompted, type **Y** and press **ENTER**.

Searchable list

It is possible to find particular apps with apt search: `apt search games`. You can also examine individual packages with apt show (for example, `apt show bsdgames`).

APT is actually a front end to the lower-level `dpkg`, which you can call to see what you have installed on the system: `dpkg -l`. Even on a fresh system, that's a large listing: we'll show you how to get useful information from such listings another time.

You can see the list of games you installed earlier with this command (scroll back to the top of the output and look for files in **/usr/games**):

```
dpkg -L bsdgames
```

Downloaded packages hang around in **/var/cache/apt** and if you find yourself short on disk space, issuing `sudo apt clean` will clear out the archive, without affecting the installed software.

Now, remember the extra details that `ls -lh` showed us in Chapter 1, *Don't Panic*? Try `ls -lh /etc/apt`. That `-rw-r--r--` at the beginning of the listing for **sources.list** comprises file attributes, telling you who may access the file and how (more on that in a moment).

Other entries in the listing have a `d` at the beginning, indicating they are directories. You'll also see hardware devices have a `c` here, for character device — `ls -l` on `/dev/input`, for example. On GNU/Linux, everything is a file, even your mouse! A dash (`-`) at the start tells us this is just a regular file; it's the remaining nine characters that cover permissions.

Every file has an owner and a group membership. Files in your home directory belong to you. If you're logged in as user pi and `ls ~ -l`, you'll see `pi pi` in each listing, telling you the owner and the group. Note that we put the switch at the end this time: that's a bad habit under certain circumstances, but we're just showing you what's possible. Owner and group aren't always the same, as `ls -l /dev` will show you.



FREE TO USE

Software in the Raspberry Pi OS repository is not just free to use, but freely modifiable and redistributable. Free software, like Raspberry Pi OS's Debian base, is built on sharing: for education and for building community.

File attributes and permissions

The file attributes, after the file type, are three groups of three characters (`rwx`) telling you which users may read, write, or execute the file or directory for, respectively, the user who owns the file, the group owner, and everyone else ('others'). Execute permissions are needed to run a file if it's a program, and for directories, so that you may `cd` into them. In the case of **/etc/apt/sources.list**:

1. The first `rw-` tells you that the owner (root) can read or write it, but the `-` in the third location means it doesn't have the execute permission set (if it did, it would be `rwx`). Because this is neither a script nor a program, but a data file, the execute permission is not needed.

2. The following `r--` tells you that members of the group that owns the file (also named root) have only read access to the file. But the owner's permissions take precedence, so the root user (which includes you, acting under the power of sudo) will always have access.
3. The final `r--` is for all other users, who have only read access.

`cd` into **/usr/lib/firefox** and then enter the command `sudo chmod a-x launcher.sh` — the `a` stands for all (user, group, and others), use `u`, `g`, or `o` to just change one. In this case, the `-` means to take away a permission. After you do this, try opening Firefox from the main menu and it won't work. You can return things to normal with `sudo chmod a+x launcher.sh`.

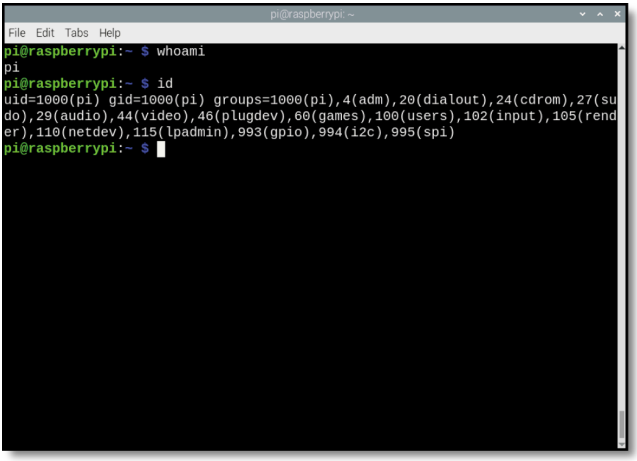


Figure 3-3 The *id* command shows what group access you have, for permission to use and alter files and devices

Octal version

As an alternative to using the `a+x` notation, we could have typed `sudo chmod 755 launcher.sh`. Those numbers are an octal representation of user, group, and others' permissions: in each case, read is represented by 4, write by 2, and execute by 1, all added together. So here we have 7s for

read+write+execute for user, and 5 for read+execute for group and all other users. `ls -l` and you'll see we're back to `-rwxr-xr-x`.

You can use `chown` to change who owns a file and `chgrp` to change which group it belongs to. Make a new text file and then run `sudo chown root myfile.txt` — now try editing it and you'll find that while you can read the file, you can no longer write to it. You can also make a file that you can write to and run, but not read!

In the next chapter, we'll be doing useful things with the output of our commands; before moving on, though, why not try some of the games from the *bsdgames* package we installed? If you'd like to try an old-school role-playing game, try running `hack`, but make sure you have a few hours free first! If you enjoy interactive fiction, `bsdgames-adventure` is an implementation of the classic Colossal Cave adventure.



Chapter 4

Manipulating text

Connect simple commands with pipes for more powerful text processing

The Unix family of operating systems, which includes the Unix-like GNU/Linux and also Apple's macOS (which is licensed to use the UNIX trademark), deals with data from commands as streams of text. This means that commands can be chained together in countless useful ways. For now, though, we'll focus on giving you a firm foundation to building your own custom commands.

Getting our feet wet

When a command is called at the terminal, it is given three streams, known as standard input (stdin), standard output (stdout), and standard error (stderr). These streams are plain text, and treated as special files. As we noted in Chapter 3, *Permission to install*, 'everything is a file': this is what gives Unix-like systems the ability to put together simple commands and programs to build complex but reliable systems.

Normally, stdin is what you enter into the terminal, while stdout (command output) and stderr (any error messages) appear together. The reason the last two have a separate existence is that you may want to redirect one of them — error messages, for example — somewhere away from the

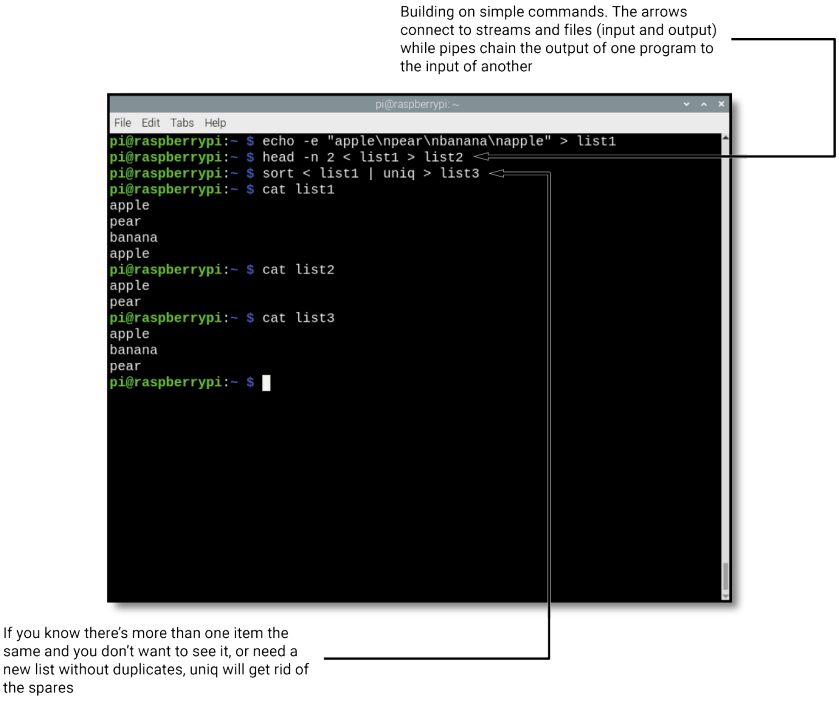


Figure 4-1 Sort and summarize files in all kinds of ways

regular output your commands produce. We'll look at separate error messages later, but first we need to know how to redirect and connect our output to other commands or files.

Connecting commands together are pipes, the | symbol found above the backslash on both GB and US keyboards (although the two keyboards for English speakers place the \ respectively to the left of **Z**, and at the far right of the home row). When you type a command such as `ls -l`, the output is sent by Raspberry Pi OS to the stdout stream, which by default is shown in your terminal. Adding a pipe connects that output to the input (stdin stream) of the next command you type. So...

```
ls -l /usr/bin | wc -l
```

...will pass the long listing of the **/usr/bin** directory to the wordcount (`wc`) program which, called with the `-l` (line) option, will tell you how many lines of output `ls` has. If the `-l` option didn't include a total amount of disk space (measured in one-kilobyte blocks) as its first line, this would be a good way to count how many files and folders are in a particular directory. However, you can get a bare listing with `ls -1` (that's the number one, not a lower-case L), and you can include hidden files, but not the current working directory (`.`) or the parent (`..`) with `ls -A1`.

Search with grep

One of the most useful commands to pass output to is `grep`, which searches for words (or 'regular expressions', which are powerful search patterns understood by a number of commands and languages), like so:

```
grep if /usr/share/code-the-classics/soccer/soccer.py
```

This displays every line in the *soccer.py* file containing the character sequence 'if' (**Figure 4-2**, overleaf) — in other words not just the word 'if', but words like 'elif' (Python's `else if`), and words like 'difficulty' if they were present.

You can use *regular expressions* to just find lines with 'if', or lines beginning with 'if', for example. The `\b` *escape character sequence* refers to a word boundary, but you need to enclose the whole expression in quotes to prevent the shell from misinterpreting it. This will find only those lines that contain the word 'if':

```
grep '\bif\b' /usr/share/code-the-classics/soccer/soccer.py
```

And this uses the `^` symbol to anchor the match to the start of the line, so it will only return lines beginning with 'if':

```
grep '^if' /usr/share/code-the-classics/soccer/soccer.py
```



REGEXP

Regular expressions (regexps) are special characters used in text searches, such as `[a-z]` to match any letter (but not numbers), `^` to match to the beginning of a line, and `$` to match the end of a line.

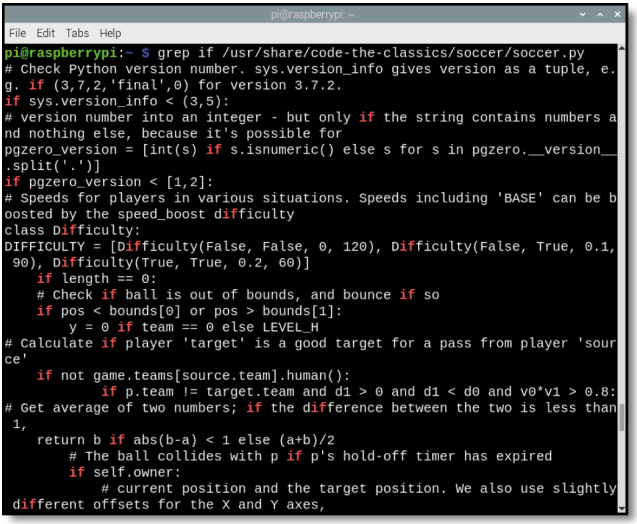


Figure 4-2 No matter how long the file, `grep` will dig out the lines you need. It's also handy for finding the results you want from a multi-page output

Piping search results and listings to `grep` is the way we find a needle in one of Raspberry Pi's haystacks. Remember `dpkg` from the last chapter, to see what was installed? Try...

```
dpkg -l | grep -i game
```

...to remind yourself which games you've installed (or are already installed). The `-i` switch makes the search *case insensitive*, as the program may be a 'Game' or 'game' in the description column. A simple `dpkg -l | more` lets you see output a page at a time.

`sort` will, as the name suggests, sort a listing into order, with various tweaks available such as `-f` to bring upper and lower case together. One way to collect unsorted data is to combine lists. `sort` will put the combined listing back in alphabetical order (`-f` indicates a case-insensitive sort):

```
ls ~ /usr/share/code-the-classics | sort -f
```

Suppose you copied one of the games to your home directory to modify: you know it's there, but you don't want to see the same name twice in the listings. `uniq` will omit the duplicated lines or, with the `-d` switch, show only those duplicates:

```
ls ~ /usr/share/code-the-classics | sort -f | uniq
```

File it away

Pipes are not the only form of redirection. `>` (the 'greater than' symbol) sends the output of a program into a text file, either creating that text file in the process, or writing over the contents of an existing one.

```
ls /usr/bin > ~/mylisting1.txt
```

> **ABSOLUTE PATH**
>
> We're using **~/mylisting1.txt** with `~` short for `/home/pi`. If you `cd` to `~` then you can simply use the file name without the **~/**

Now look in **/home/pi/mylisting1.txt** and you'll see the output of `ls /usr/bin`. Note that each item is on a separate line (**Figure 4-3**). Your terminal displays multiple listings per line for space efficiency; however, for easy compatibility between commands, one listing per line is used. Most commands operate on lines of text; e.g., `grep` showed you in which lines it found 'if'. Note that some commands need a dash as a placeholder for the stdin stream being piped to them:

```
echo "zzzz is not a real program here" | cat ~/mylisting1.txt -
```

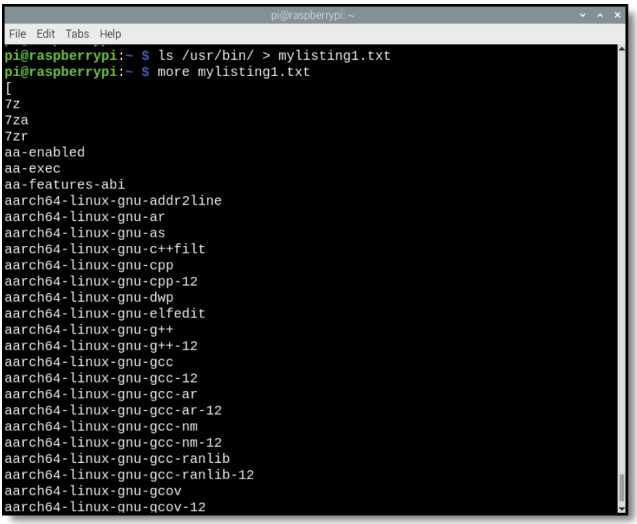


Figure 4-3 With redirection, you can get all of the output from a command saved straight into a text file. Save your error messages to ask about them on the forums!

Appending

If you want to add something to the end of a file without overwriting the contents, you need `>>`.

```
echo "& one more for luck!" >> ~/mylisting1.txt
```

`echo` simply displays whatever is in the quote marks to stdout; the `-e` switch lets you add in special characters, like `\n` for newline (see below). You can look at the last few lines of a file with `tail ~/mylisting1.txt`. `<` will link a program's input stream to the contents of a file or stream. Make an unsorted list to work on, and sort it:

```
echo -e "aardvark\nplatypus\njellyfish\naardvark" > list1
sort < list1
```

You can also combine `<` and `>`:

```
head -n 2 < list1 > list2
```

...will read from **list1**, passing it to **head** to take the first two lines, then putting these in a file called **list2**. Add in a pipe:

```
sort < list1 | uniq > list3
```

Lastly, let's separate that stderr stream: it has file descriptor 2 (don't worry too much about this), and **2>** sends the error messages to any file you choose:

```
cat list1 list2 list3 list42 2>errors.txt
```

The screen will display the 'list' files you do have, and the 'No such file or directory' message(s) will end up in **errors.txt** — **2>>** will append the messages to the file without overwriting previous contents.

FILING HOMEWORK



There are many more commands beyond **grep**, **sort**, and **uniq** that can be chained together. Take a look at **cut** if you're feeling adventurous.



Chapter 5

Customise the command line

Make Raspberry Pi OS a little more personal as you get it to behave and look just the way we want it — for yourself and other users

Take a look at that blinking cursor on your terminal: `pi@raspberrypi ~ $`. The $ is known as the 'dollar prompt', awaiting your command; before it you see the **~** (tilde), shorthand for 'home' — which is **/home/pi** in our case (yours is probably different). Before that is `[user name]@[computer name]`, in the form *pi@raspberrypi*. Not only is this informative (at least if you've forgotten who and where you are), but it's also something you can change and personalise.

New user

If more than one person uses your Raspberry Pi, you can set up user accounts for each person. Creating a new user in Raspberry Pi OS is easy: `sudo adduser babbage` will create a new user account named *babbage*. You will be prompted for a password (pick a good one) and lots of irrelevant info (dating back to shared university computers of the 1970s) that you can safely ignore by pressing **ENTER** at each prompt. Now that we

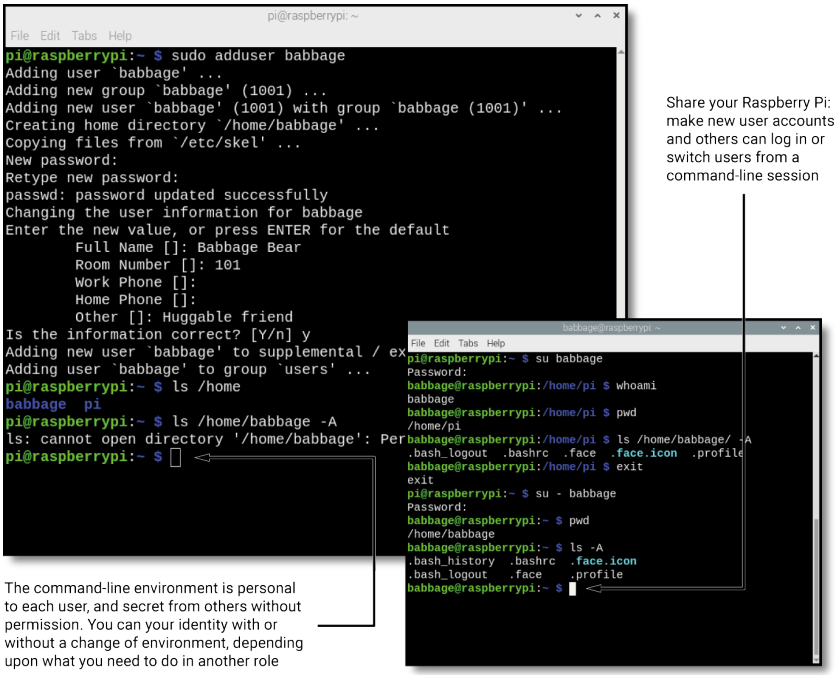


Figure 5-1 It's your world

have a user account for babbage, have a look at their home directory with `sudo ls -A /home/babbage`. As babbage has never logged into the computer, you will see it's missing the directories you have in **/home/pi** (for now), but there are a few config files. The first time you log into the desktop as babbage, Raspberry Pi OS creates the missing folders.

If you want your Raspberry Pi to prompt for a username and password when it starts up, run `sudo raspi-config`, go into System Options, choose Auto Login, answer no to both questions (console and desktop autologin), choose Finish, and reboot.

Not every user has a home directory and logs in: run `cat /etc/passwd` and you'll see many users that aren't people. This is because files and running programs on Unix-like systems must belong to a user (and a group —

see **/etc/group**), as we saw in Chapter 1, *Don't Panic* when we ran `ls -l`. If you want to change a password, you'll need to use the `passwd` command: `sudo passwd babbage` changes the password for babbage. Running `passwd` without a username prompts you to change your password.

Transformations in the virtual world are always easier than those in nature, and this is the case with switching from being 'pi' to 'babbage': we use the change (or substitute) user command, `su`, like so: `su babbage`. Supply babbage's password, and you should see the prompt change to `babbage@raspberrypi`; you can confirm who you are with `whoami`.

HOME RUN

If you're logged in as user *pi*, then `~` is a shortcut to **/home/pi** — but `~babbage` can be used as a shortcut for **/home/babbage**.



Changing identity

`su - babbage` (note the dash) is best, as you'll gain all of babbage's environment settings, and will change directory to **/home/babbage.**

On older Unix-like systems, running `su` without a username lets you become the root user, with absolute powers (permissions to run, edit, or delete anything). Modern systems like Raspberry Pi OS prefer `sudo` to run individual programs with root permissions. Root's powers may be temporarily attained with `sudo -s` — try it (as *your* username, because babbage won't have access to `sudo`; run `man visudo` to learn how to change this) and note how the prompt changes (type `exit` to exit) — but it's a bad idea to run with more permissions than you need!

You can customise elements of a user's shell by editing their **~/.bashrc**. Take a look through that configuration file now (as user babbage): `more ~/.bashrc`. Note a number of variables in all capital letters, such as *HISTSIZE* and *PS1*. The last of these controls the prompt you see, such as `babbage@raspberrypi ~ $`. To change it for the duration of your current session, try something like: `export PS1="tutorial@rpom > "`.

This is a temporary change: type **exit** and you've left the **su** value of *babbage*, so you'll see **pi@raspberrypi ~ $** once more. If you **su** back to babbage, the rpom prompt will still be gone. To make your change permanent, you need to put the *PS1* value you want into **~/.bashrc**. A web search will bring up many options for better customising your prompt.

Bash stores information, from your previous working directory to who you are, in environment variables like OLDPWD and USER. See individual variables with e.g. **echo $USER**, or view them all with **env** (see **Figure 5-2**).

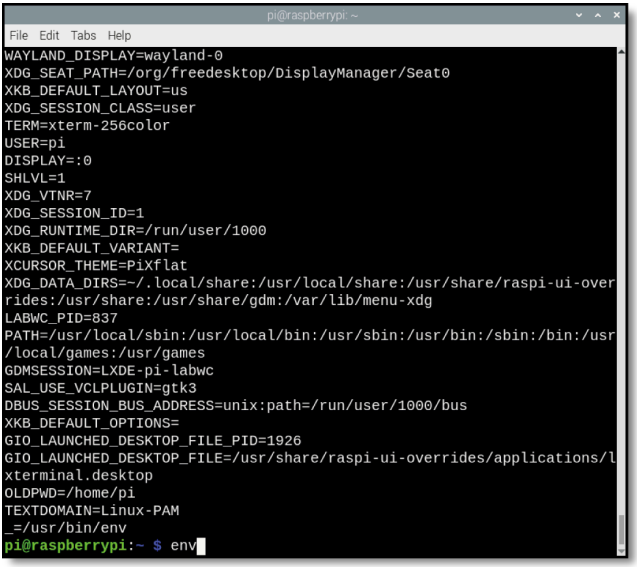


Figure 5-2 Viewing all the variables

The **~/.bashrc** file is read each time the shell starts, or in other words, every time you log into a console or open a terminal. That's unless you change the default shell in Raspberry Pi OS away from Bash, something you may have reason to do in the future — there are many alternatives available — but let's not worry about that for now. You can put all sorts of commands in **~/.bashrc** to personalise your environment: command aliases are great for regularly used combinations.

WHO AM I?

From the terminal or a virtual console (**CTRL+ALT+F1** to **F6**), you can `su` to someone else, but start an app from the desktop menu and it will run under whichever user you logged in as.

Aliases

Aliases let you change the default command-line arguments to programs. As user pi, you can see the defaults with `grep alias ~/.bashrc`. There are a few aliases already in there, particularly for the `ls` command. One entry is: `# alias ll='ls -l'`. This sounds quite useful, although the `#` indicates that it is 'commented out', which means that it will not be read by Bash. Open **.bashrc** in your text editor (in File Manager, press **CTRL+H** to show hidden files, then double-click **.bashrc**) — the simple Text Editor will do for now (we touched on editing text files with nano earlier, and we'll discuss it further in "Editing with nano" on page 44). Remove the `#`, open a new terminal, and try typing `ll` to run `ls -l`. Handy, but it could be better. Change it to: `alias ll='ls -lAhF'` to get output in kB or MB, rather than bytes, along with trailing slashes on directory names, and the omission of the ever-present . and **..** (current and parent) directories.

You can make `rm`, `cp`, and `mv` much safer with these aliases (`-i` requires confirmation when overwriting or deleting files):

```
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
```

Changes to **.bashrc** take effect in your next terminal session, but you could run the `alias` command in your current session. To disable an alias for a session, use: `unalias ll`.



Chapter 6

Connecting disks

Disk drives come and disk drives go — tackling the management of removable storage

Although Raspberry Pi OS will, when running the GUI desktop, automatically mount any disk-type device (USB flash key, camera, etc.) plugged into the USB port and offer to open it, you may wish to get more direct control of the process. Or, as is more often the case, you may want to mount a disk when the Raspberry Pi is running a project that doesn't involve booting as far as the GUI, as it's not necessary for most sensor projects, robots, etc. This chapter has plenty of new concepts to get to grips with — but they'll also be useful when a USB drive fails to mount, and the GUI doesn't give you enough useful information about what's going on.

IN DEPTH

If you want to delve deeper into what goes on inside Raspberry Pi OS and other Unix-like systems, try Brian Ward's excellent book, *How Linux Works* (**rpimag.co/howlinuxworks**).

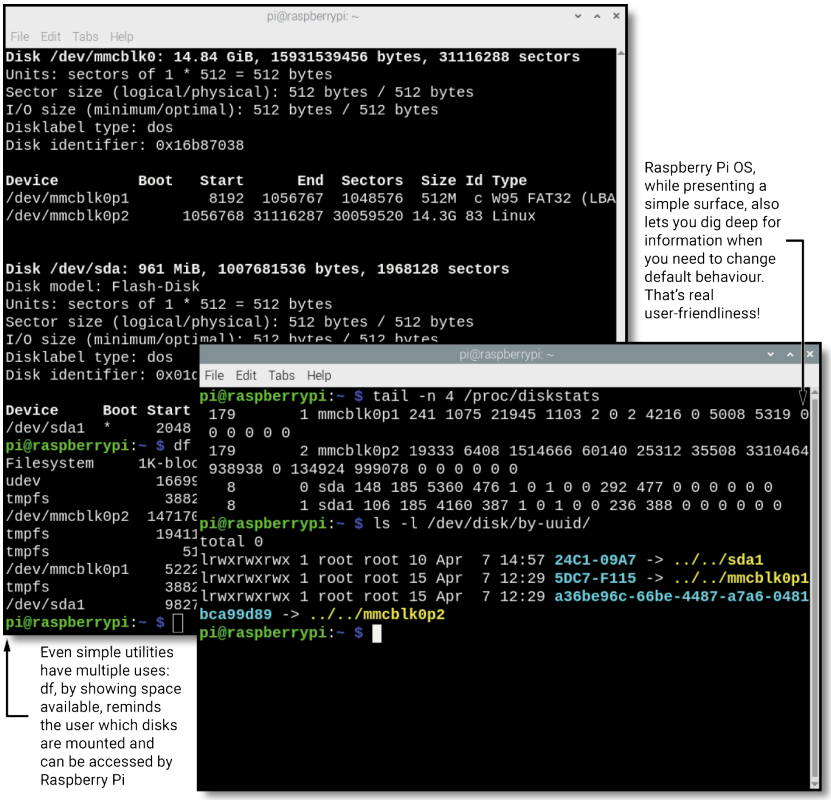


Figure 6-1 Working with disks from the command line

Connected or mounted?

Raspberry Pi OS wants to mount plugged-in disks and take care of the details for you — note how in **Figure 6-2**, the GUI tells you it's a 'removable disk' — while the command line beneath has information for you to take control when you need the job done in a particular way, telling you it's an FAT file system.

Plugging a drive or flash memory device into your Raspberry Pi (connecting it) is not the same as making it available for Raspberry Pi to interact

with (mounting it) — knowing what's on it and reading, writing, and altering files there. It's an odd concept to accept: the computer knows there's a disk plugged in, but its contents remain invisible until Raspberry Pi is told to mount it (by you or by the desktop GUI). It's like seeing a book on your shelf, but not being allowed to open or read it.

Disks and disk-like devices are mounted by Raspberry Pi OS on a virtual file system, and you'll rarely need to worry about what goes on beneath that layer of abstraction, but to see some of it, type `mount`. The information displayed is of the form device on mount point, file-system type, options. You'll see lots of device without the prefix **/dev** for parts of the virtual system that you don't need to worry about; the devices that concern us start with **/dev** and have names like **/dev/mmcblk0p1** for partitions of the Raspberry Pi's SD card, and **/dev/sda1** for plugged-in USB drives.

Plug in a USB drive (a flash drive should work fine, but some hard drives may require a separate power supply). Like most computer desktops, Raspberry Pi OS automatically mounts the disks, so (unless you boot to a virtual console rather than the desktop) you'll need to unmount it. `mount` will show an entry beginning something like `/dev/sda1 on /media/pi/UNTITLED…` and you can unmount with `sudo umount /dev/sda1` (yes, that is umount without an 'n' after the first 'u'). An error will result if the device is in use, so change directory and/or close apps using files from the device. Now we can mount it just the way we want.

Finding the disk

The **/dev/sda1** part refers to the first and only partition (a section of the hard drive that is separated from other segments) on **/dev/sda**. The next device plugged in would be assigned **/dev/sdb1**. You can see what's being assigned by running `tail -f /var/log/messages`, then plugging in the USB device. On other systems, if **/var/log/messages** draws a blank, try **/var/log/syslog**. Stop the tail with **CTRL+C**. Another way of seeing connected devices that aren't necessarily mounted is `fdisk`, a low-level tool used to divide disks up into partitions, before creating file systems on those disks. Called with the list option, `sudo fdisk -l`, it performs no partitioning, but simply lists partitions on those disks connected to Raspberry Pi. It also gives file-system information, which you need in order to mount the disk.

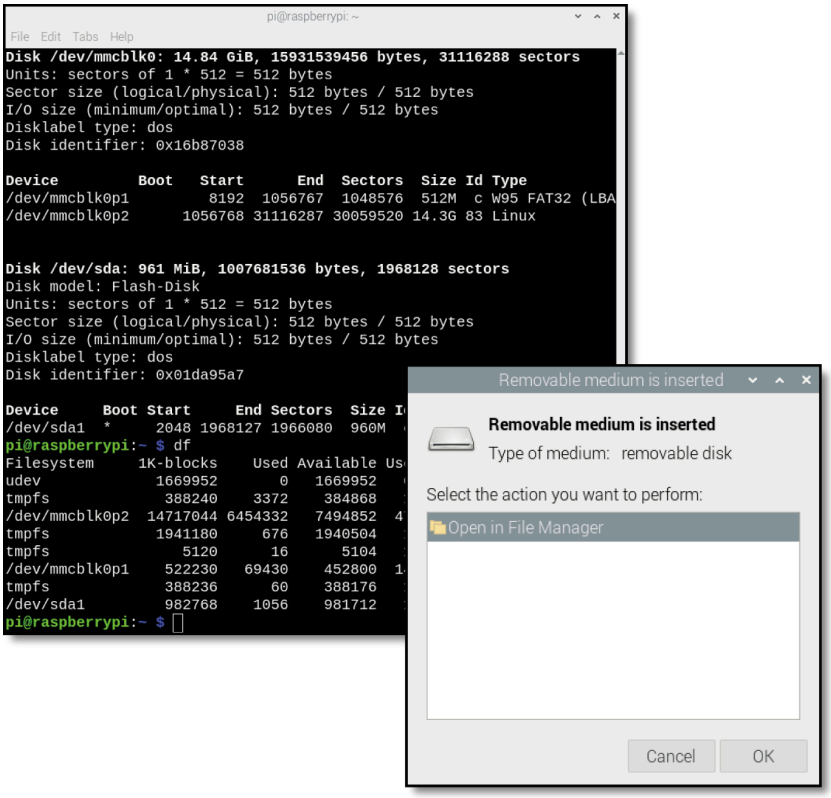


Figure 6-2 Raspberry Pi OS prompts you to mount a disk



DISKS WITHOUT PARTITIONS

Some USB sticks are *partitionless*, so if the output of **`mount`** shows no partition number (for example, **/dev/sda** rather than **/dev/sda1**) or the output of **`fdisk`** looks strange (for example, a 1GB drive shows multiple 1TB partitions), you can use **`blkid`** (discussed in the next section) to get information about the disk, including its file-system type.

Before you proceed, you need a mount point (somewhere to place the device on the file-system hierarchy) with appropriate permissions. Create one with:

```
sudo mkdir /mnt/usb
sudo chmod 775 /mnt/usb
```

Mount the disk with `sudo mount -t vfat /dev/sda1 /mnt/usb`, where vfat (or exfat, NTFS, ext3, etc.) is the file-system type you got from `fdisk` or `blkid`. Remember to replace `sda1` with the partition or disk you want to mount.

> **FORMAT**
>
> Should you need to format a new partition on Raspberry Pi OS, read the manual: `man mkfs` and for individual file-system types such as `man mkfs.ext4`.



File-system table

Raspberry Pi OS knows which disks to mount at boot time by reading the file-system table (**/etc/fstab**), and we could put our **/dev/sda1** in there, but if we start up with two drives plugged in, the wrong one may be selected. Fortunately, disks (or rather, disk partitions) have unique labels known as UUIDs randomly allocated when the partition is created. Find them with `sudo blkid`, which also helpfully tells you the label, if any, that often contains the make and model of external drives, or look in **/dev/disk/by-uuid**.

For our VFAT-formatted drive, we ran `sudo nano /etc/fstab` and added the following line to the end of the file:

```
/dev/disk/by-uuid/2A0F-AF03 /mnt/usb1GB vfat nofail 0 0
```

This specifies the device name (yours will be different), mount point, file-system type, option, and two numeric fields: the first of these should be zero (it relates to the unused dump backup program), while the second is the order of check and repair at boot: 1 for the root file system, 2 for other

permanently mounted disks for data, and 0 (no check) for all others. `man mount` will tell you about possible options. The `nofail` option will ensure that your Raspberry Pi will boot even if the USB drive is not present.



DISK & DISK SPACE

The `df` command shows you space on mounted drives: just type `df` and you'll also get a list of connected drives. It's more readable than `mount`, though lacking file-system type info. It's also quicker to type!

Editing with nano

We touched briefly on nano in Chapter 2, *Read and write text*. Looking in a little more depth, the first thing to be aware of is the dozen shortcuts listed across the bottom two lines of the terminal: each is the **CTRL** key (represented by the caret **^**) held at the same time as a single letter key. For example, **^R** for ReadFile (i.e. open), **^O** for WriteOut (in other words, *save*), and **^X** for Exit. Remember those last two for now, and you'll be able to manage nano. However, if you learn more of them, you will really race through your editing tasks.

While nano lacks the power features of Emacs and Vim, its two main command-line code editor rivals, it has useful features such as a powerful Justify (**^J**), which will reassemble a paragraph of line-break strewn text into an unbroken paragraph, or apply line breaks at a fixed character length. This is a legacy of its development for email composition. **^K** cuts the line of text the cursor is on, but it isn't just a delete function: each cut is added to a clipboard. **^U** will paste the entire clipboard at the cursor position: it's great for gathering together useful snippets from a longer text.

Hit **^O** to save **fstab**, and the shortcut listing changes, with many now prefixed with M instead of **^** — the M is short for Meta, which means the **ALT** key on your keyboard (once upon a time, some computers had several modifier keys, such as Super and Hyper). One 'hidden' shortcut after **^O** is that at this point, **^T** now opens a file manager to search for the file/directory where you want to save. For now, don't explore any of those shortcuts: just save the file.

After saving, exit nano and run the command `systemctl daemon-reload` to reload the fstab; now `sudo mount -a` will mount the external drive at the desired mount point (**Figure 6-3**), regardless of what else is plugged in. If you have other new entries in **/etc/fstab**, then `sudo mount /mnt/usb1GB` (or whatever entry you put in **fstab**) will mount just that chosen device if you don't want to mount any of the others. What's more, next time you reboot your Raspberry Pi, the drive will be mounted if it's plugged in.

Having got inside connected disks, the next chapter will see us accessing all of Raspberry Pi, but remotely, from anywhere on the planet with an internet connection.

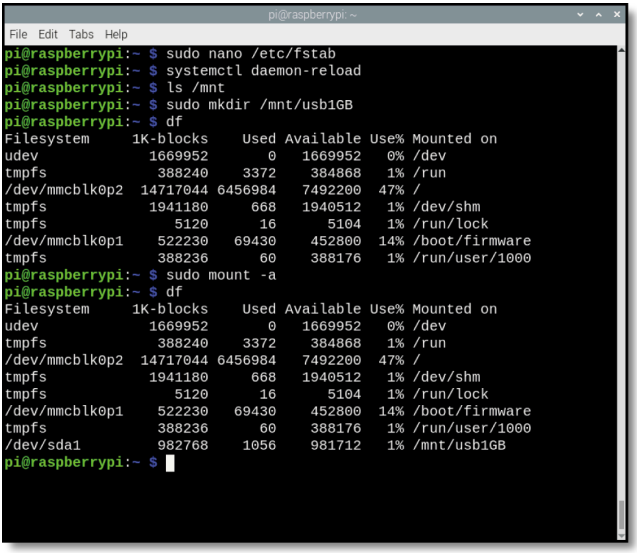


Figure 6-3 `mount -a` reads **/etc/fstab** to mount your disks



Chapter 7

Predictable Networking

In this chapter, we give the Raspberry Pi a permanent network address of its own

Raspberry Pi OS takes care of automatically connecting in most situations, but sometimes you need to override automatic configurations to ensure a consistent network setting for your Raspberry Pi project: Raspberry Pi OS has the tools, and we'll show you the essentials you need to stay connected.

Plug an Ethernet cable from your router to your Raspberry Pi (or connect via WiFi) and, automatically, Raspberry Pi OS knows where it is on the network, and can talk to the outside world.

All of this is thanks to DHCP — Dynamic Host Configuration Protocol — which provides network configuration for every device connected into a network. Typically, this comes in the form of an IPv4 (Internet Protocol version 4) address, a pair of four numbers separated by a period. For example: 192.168.1.100

The first two sets of numbers, 192.168, mark the start of a private range. These are the numbers for all devices in your house, ranging from 192.168.0.0 to 192.168.255.255.

Check your Raspberry Pi's current connection with the **`ifconfig`** command. This should show, amongst others, a line like **`inet 192.168.1.100`**

(with your own numbers). This will be below the **eth0** section if you are connected via Ethernet, or under the **wlan0** section if you're connected via WiFi.

A faster way to get your IP address is to run the command **hostname -I**.

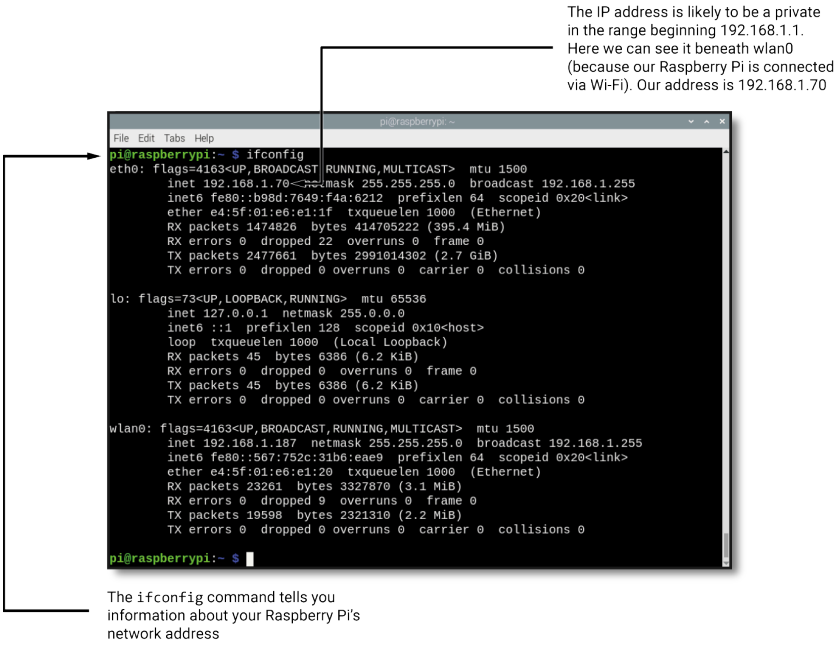


Figure 7-1 Examining your network configuration

Configure your Raspberry Pi with a static IP address

Usually when you connect a Raspberry Pi to a local area network (LAN), it is automatically assigned an IP address. This address may change the next time you reconnect to the network.

Sometimes, however, you might want your Raspberry Pi to boot up with the same IP address each time. This can be useful if you are making a

small self-contained network, or building a standalone project such as a robot. Here's how to do it.

Raspberry Pi OS uses a system called NetworkManager to manage network connections. You can use the `nmcli` tool to configure these connections. Run the command `sudo nmcli dev status` to see a list of connections. You'll see output similar to this:

```
DEVICE     TYPE      STATE                   CONNECTION
eth0       ethernet  connected               Wired connection 1
lo         loopback  connected (externally)  lo
wlan0      wifi      connected               preconfigured
p2p-dev…   wifi-p2p  disconnected            --
```

You can determine the name of the connection to edit by checking its type. In the preceding example, we have one Ethernet, one WiFi, and one loopback (for the localhost, which we'll discuss later in this chapter). You don't want to mess with the loopback settings, and you can ignore wifi-p2p. Suppose you want to set the Ethernet connection to a static IP. You could type the following (the `\` symbol allows us to break a single command across two lines):

```
sudo nmcli con mod "Wired connection 1" \
  ipv4.addresses 192.168.1.253/24 ipv4.gateway 192.168.1.1 \
  ipv4.method manual
```

You should replace `192.168.1.253` with the IP address your Raspberry Pi currently has (or the one you want it to have), and `192.168.1.1` with your network gateway (usually your router) IP address. If your network is configured with anything other than a 24-bit network prefix (in this case, 192.168.1.x), you'll need to change the /24 as appropriate.

Your Raspberry Pi will now boot up with the specified IP address every time; we didn't use 192.168.1.1 as this is reserved for the router. You can of course use any address you like, but in the configuration above, the range must be between 192.168.1.2 and 192.168.1.254, and may depend on your router and other devices connected to the network.

Reboot with `sudo shutdown -r now`. Log in and type `hostname -I`. You should then see the IP address you set.

Normally you don't want your computer set to use a static IP address. You can change the network configuration back by running the following command, and then rebooting again:

```
sudo nmcli con mod "Wired connection 1" \
  ipv4.addresses "" ipv4.gateway "" ipv4.method auto
```

If you don't want to reboot between making changes, you can use the command `sudo nmcli dev reapply eth0` to immediately apply the changes (note that we use the device name rather than the connection name here).

Ping!

Ping is the most basic tool in the network testing armoury, but it is one which is often called upon. It sends an ICMP (Internet Control Message Protocol) ECHO_REQUEST to a device on the network. ICMP is built into every connected device and used for diagnostics and error messages: a ping will produce a reply from the pinged machine, which tells you it is on, and connected, and that the network is working between you and it. Information about packets lost, and time taken, also helps with fault diagnosis.

A successful `ping localhost` from Raspberry Pi tells you not just that the local loopback interface is working, but that localhost resolves to 127.0.0.1, the local loopback address. Name resolution is the cause of many computing problems. Try to ping your Raspberry Pi from another machine on your local network: `ping 192.168.1.100` (**Figure 7-2**) — you'll need to use your actual IP address though. If you're doing this from a Windows machine, ping defaults to four attempts; from another system (another Raspberry Pi, a Mac, or Ubuntu or other GNU/Linux), it will carry on until you stop it with **CTRL+C**, unless you set a number of ECHO_REQUEST sends with `-c` like so (this will attempt to ping Raspberry Pi's web server):

```
ping -c 5 raspberrypi.com
```

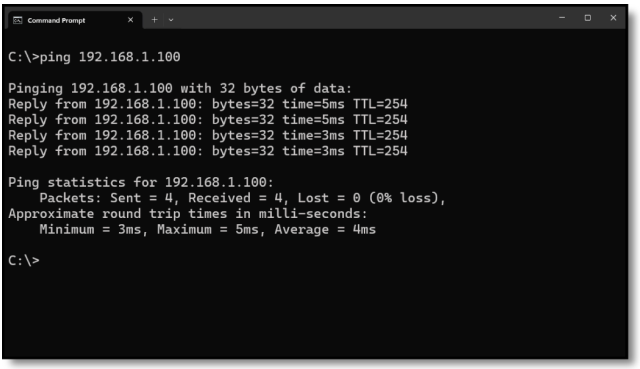


Figure 7-2 Use ping from another computer to detect if your Raspberry Pi is responding to network requests

IPv6

The four-digit IP address style we use (such as 192.168.1.100) is IPv4. The newer standard, IPv6, is becoming more common. These are longer 128-bit addresses represented in hexadecimal (for example, fd51:42f8:caae:d92e::1).



Chapter 8

Stopping a process

As close to perfect as Raspberry Pi OS is, things can go wrong. In this chapter, we learn that there's no need to turn Raspberry Pi off and on again: just kill the process!

Ever lost the 'off switch' for a program? Sometimes a piece of software you're running seems to have no inclination to stop: either you cannot find how to quit, or the app has a problem, and won't respond to your **Q**, **CTRL+C**, or whatever command should close it down.

There's no need to panic, and certainly no need to reboot; just identify the process and quietly 'kill' it. We'll show you how, and we'll look at what else can be done with knowledge of processes.

Processes

Find the many processes running on your Raspberry Pi with `ps` (**Figure 8-2**). As a minimum, you should run it with the `a` and `x` options — which together give all processes, rather than just those belonging to you or just those attached to a terminal — and with `u` to see a user-oriented output format that includes detailed process information; `w` adds wider output, and `ww` will wrap over the line end to display information without truncating the output.

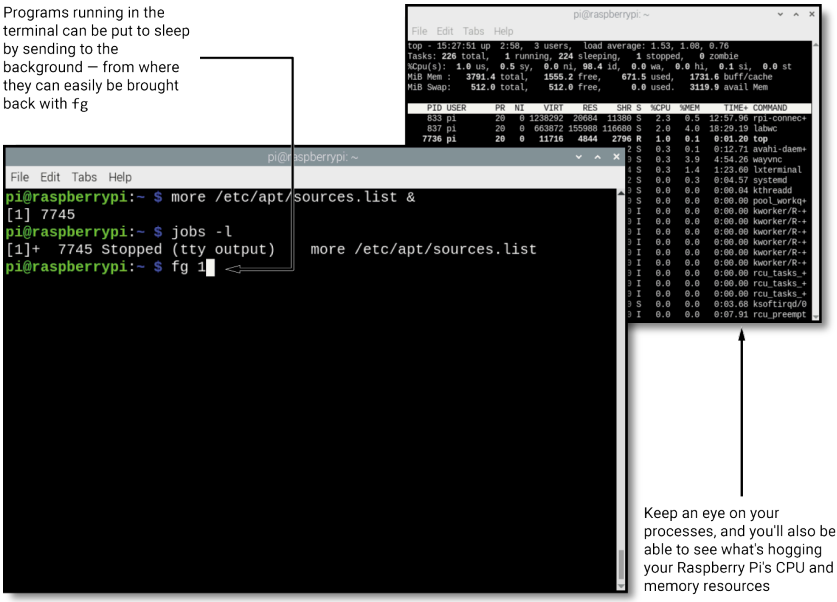


Figure 8-1 Working with processes



BOOT SYSTEM

The startup process of Raspberry Pi OS is controlled by systemd, like other GNU/Linux distributions. It's fairly new in Linux terms — systemd came out in 2010, when Linux was just 19 years old (and UNIX was 41).

Type `ps auxww` to see, then try with just `a` or other combinations. You will notice that these options work without the leading dash seen for other commands. Both the lack of dashes, and the particular letters, `a` and `x`, date back to the original UNIX ps of the early 1970s, maintained through various revisions by one of UNIX's two family branches, BSD, and baked into the first GNU/Linux ps. UNIX's other branch, System V, had extended and changed ps with new options and new abbreviations for command switches, so for `ps ax` you may see elsewhere `ps -e` (or `-ef` or `-ely` to

show in long format). The **ps** that comes with Raspberry Pi OS will work with both BSD and System V command-line arguments.

The **ps aux** listing has various headers, including the USER which owns the process, and the PID, or Process IDentification number. The list starts with 1 for **init**, the parent process of everything that happens in user-space after the Linux kernel starts up when you switch Raspberry Pi on.

Knowing the PID makes it easy to kill a process, should that be the preferred way of shutting it down. For example, to kill a program with a PID of 3012, simply enter **kill 3012**, and to quickly find the process in the first place, use **grep** on the **ps** list. For example, locating vi processes:

```
ps aux | grep -i vi
```

The **-i** (ignore case) isn't usually necessary, but occasionally a program may break convention and contain upper-case letters in its file name. You can also use **killall** to kill by program name: **killall firefox**

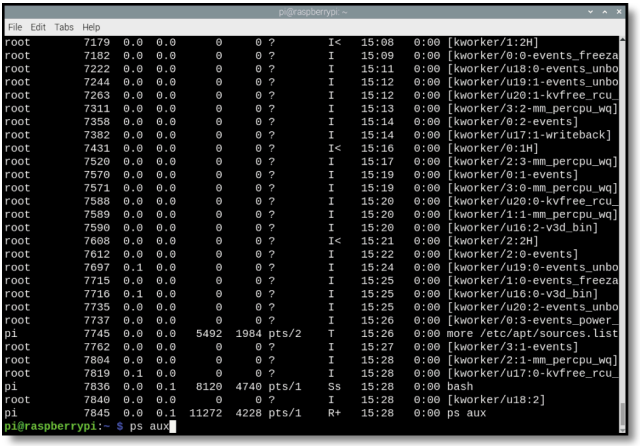


Figure 8-2 Everything running has a process ID (PID) that can be used to control that program; find them all with **ps aux**

Piping commands

You can pipe `ps`'s output to select the PID and feed directly to the `kill` command:

```
kill $(ps aux | grep '[f]irefox' | awk '{print $2}')
```

We don't have space for an in-depth look at `awk` (we're using it here to print the second field of grep's output: the PID), but the `[f]` trick at the beginning of 'firefox' (or whatever named process you want to kill) prevents the grep process itself being listed in the results; in the earlier vi example, grep found the grep process itself as well as vi (and anything with the letter sequence vi in its name). If you run the command between `$(` and `)` by itself, you'll see the PID of Firefox displayed (if it's running). Everything between the `$(` and `)` is replaced by its output, so it's as if you typed kill followed by the PID of Firefox.

That's a fun trick, but the `pgrep` command can find the PID for you, just using the name of the process: `pgrep firefox`. The `pkill` command will kill all processes matching the name you provide: `pkill firefox`. But don't go killing processes indiscriminately; you could inadvertently terminate one that is essential to the proper functioning of Raspberry Pi OS!

The output of `ps` also shows you useful information like percentage of memory and CPU time used, but it's more useful to see these changing in real time. For this, use `top`, which also shows total CPU and memory use in the header lines, the latter in the format that you can also find by issuing the command `free`. For an improved top, you can use `htop` — if it needs installing, use:

```
sudo apt install htop
```

`htop` is scrollable, both horizontally and vertically, and allows you to issue commands (such as `k` for kill) to highlighted processes. When you've finished, both `top` and `htop` are exited with Q, although in `htop` you may care to practise by highlighting the `htop` process and killing it from there (see **Figure 8-3**). `htop` also shows load over the separate cores of the processor if you have a Raspberry Pi 2 or later.

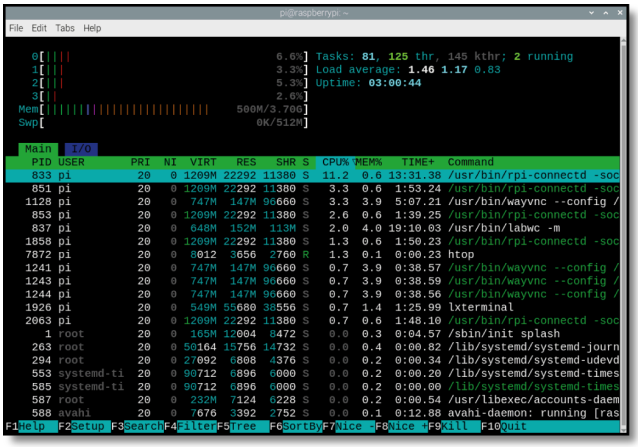


Figure 8-3 htop tells you what's running, what resources it's using, and lets you interact with the process, even killing htop from within htop

KEEP ON TOP

When using a virtual console, it can be worth keeping `htop` running so that if there are any problems, you can **CTRL+ALT-FN** (where FN is one of the F keys) there for a quick look for any problems — even if the GUI freezes.

Background job

Placing an ampersand (**&**) after a command in the shell will place the program in the background — try with: `man top &` and you'll get an output like: *[1] 12768*.

The first number is a job number, assigned by the shell, and the second the PID we've been working with above. `man top` is now running in the background, and you can use the job control number to work with the process in the shell. Start some other processes in the background if you wish (by appending **&**), then bring the first — `man top` — to the foreground with `fg 1`. Now you should see `man` running again.

You can place a running shell program in the background by 'suspending' it with **CTRL+Z**. `fg` will always bring back the most recently suspended or backgrounded job, unless a job number is specified. Note that these job numbers apply only within the shell where the process started. Type `jobs` to see background processes; `jobs -l` adds in process IDs (PID) to the listing.

Signals

When we send a signal from `htop`, we are given a choice of signal to send. The most important are **SIGTERM**, **SIGINT**, and **SIGKILL**.

The first is also the signal `kill` will send from the command line if not called with a modifier: it tells a process to stop, and most programs will respond by catching the signal, and first saving any data they need to save and releasing system resources before quitting.

`kill -2` sends SIGINT, which is equivalent to stopping a program from the terminal with **CTRL+C**: you could lose data. Most drastic is `kill -9` to send SIGKILL, telling the kernel to let the process go with no warning. Save this one for when nothing else works.

Mildest of all is the Hang Up (HUP) signal, called with `kill -1` (the number 1), which many daemons are programmed to treat as a call to simply re-read their configuration files and carry on running. It's certainly the safest signal to send on a critical machine. To see a list of all signals, run `kill -l` (that's a lowercase L).

Staying on

`nohup` will run a program which will continue after the terminal from which it is started has closed, ignoring the consequent SIGHUP (hangup) signal. As the process is detached from the terminal, error messages and output are sent to the file **nohup.out** in whichever directory you were in when you started the process. You can redirect it — as we did in Chapter 4, *Manipulating text* — with `1>` for stdout and `2>` for stderr; `&>` is a special case for redirecting both stdout and stderr:

```
nohup myprog &> backgroundoutput.txt &
```

One use of **nohup** is to be able to set something in motion from an SSH session, which will continue after an interruption. For example, restarting the network connection to which you are connected:

```
sudo nohup sh -c "ifconfig wlan0 down && ifconfig wlan0 up"
```

Note that you'll need **sudo** privileges to read the **nohup.out** log file created here — or you can reassign ownership with:

```
sudo chown pi:pi nohup.out
```

KEEP ON RUNNING



nohup is useful for a program that will be running for some time in the background — perhaps a sensor project you are working on — until you feel happy enough to add it to Raspberry Pi OS startup processes.



Chapter 9

Remote Raspberry Pi

Learn how to access your Raspberry Pi from remote PCs and devices with Secure Shell (SSH)

It's great that Raspberry Pi is so portable, but sometimes you may want to use it without taking it with you. Here, Raspberry Pi OS is a real strength, as Unix-like operating systems have been used this way for over 50 years.

Over time, as the internet has given the opportunity for malicious users to connect to computers, old standards like Telnet and rlogin have been replaced by Secure Shell (SSH), based on public-key cryptography. Once set up, secure connections are simple, and open to scripted, automatic connection for your projects. As a reminder, please set a secure password on your Raspberry Pi before accessing SSH.

If the SSH server is not enabled by default on your version of Raspberry Pi OS, run `sudo raspi-config` and go to the Interface Options settings to enable SSH. Now you can try connecting to your Raspberry Pi from another computer on your network.

Connecting with SSH

From a Mac or GNU/Linux computer, use `ssh` from a terminal to connect to Raspberry Pi. You can use the command-line OpenSSH client on Windows

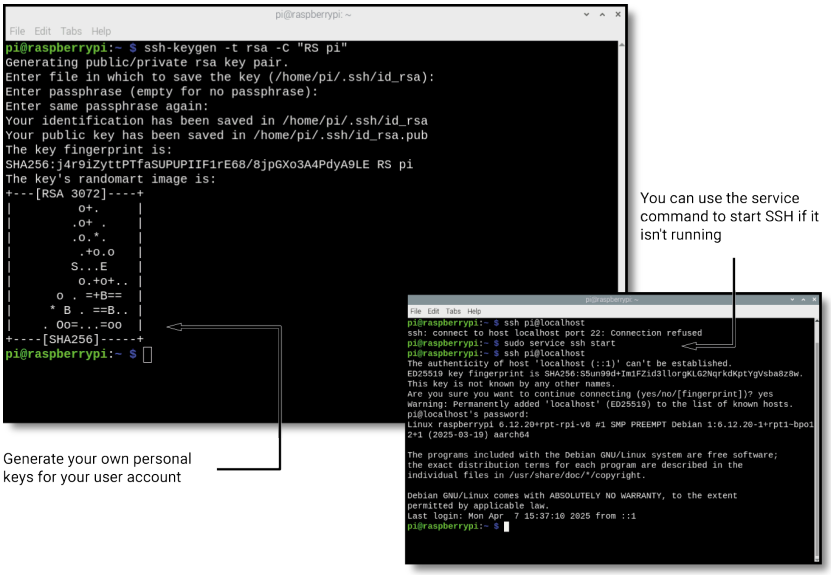


Figure 9-1 SSH and its keys

10/11 PCs; for earlier PCs, install an SSH client like PuTTY (**putty.org**), which also works with SCP, Telnet, and rlogin, and can connect to a serial port. Android users have a variety of terminal emulators on the Play Store they can use too.

You can connect with the command **`ssh [user name]@[computer name]`**, for example **`ssh pi@raspberrypi`** and enter your password (replace **`pi`** with your username and **`raspberrypi`** with your host name). The first time you connect, you'll be prompted to confirm the authenticity of your Raspberry Pi. In some cases, you may need to replace the host name with your Raspberry Pi's IP address. You can find the IP address assigned to Raspberry Pi with **`ifconfig`** (note the 'inet addr' for the eth0 or wlan0 interface) or **`hostname -I`**.

You should now be at the command-line interface of your Raspberry Pi. If you got any sort of error, check from Raspberry Pi that SSH is really up and running by entering **`ssh@localhost`** on Raspberry Pi itself. If that

works, SSH is up and running on Raspberry Pi, so take a closer look at network settings at both ends to diagnose the problem.

Hello, World

Now that you can access Raspberry Pi on the local network, it's time to share with the world. This is very easy to set up — and extremely secure — thanks to Raspberry Pi Connect, which allows you to not only connect via screen-share with any running Raspberry Pi, but also use a remote command-line interface just like SSH. Set up an account over at **rpimag.co/connect** and then on your Raspberry Pi desktop, click the Raspberry Pi Connect icon in the top right — it looks like two small squares connected via a circle.

From there, you can sign into your Raspberry Pi Connect account from your Raspberry Pi, adding it to the devices you can access from the web. When this Raspberry Pi is on, you'll find it listed on your Connect account with a 'Connect via' button for it. Click on that and select Remote Shell to access the command line from anywhere else in the world.

INTERRUPTED SERVICE



While you can restart most services with `sudo service ssh restart`, replacing `restart` with `reload` permits configuration changes to be registered with less disruption, which is key for some projects.

Bye bye FTP

File Transfer Protocol (FTP) was not designed for security: data, and even passwords, are transmitted unencrypted. The Secure Copy Program (SCP), which runs over SSH, is best for transferring files. The syntax of the command mimics the command-line cp program:

```
scp pi@raspberrypi:/home/pi/testfile1 .
```

Here we're transferring a file from Raspberry Pi, across a local network, to the current location (the dot shortcut). Note that you can use wildcards for groups of similarly named files, and can recursively copy directories and their contents with the `-r` switch after `scp`.

A secure key

You can use a personal key to skip the password prompt when using SSH to connect to your Raspberry Pi from another computer. The first step is to create a private/public key pair.

On the remote computer (the one you will use to connect to your Raspberry Pi), open a terminal or command prompt and run `ssh-keygen -t rsa -C "comment"`, where `"comment"` is anything you want to identify the key with: name, email, or machine and project, for example. When asked where to save it, accept the default.

You'll be asked for a passphrase to secure the key — if you press **ENTER**, you'll get a key with no passphrase, which is not ideal. That means that anyone who possesses the private portion of the key pair can log in without a password. If you already have a private/public key pair that you use to log into other computers, you can use that instead of creating a new one.

If you accepted the defaults, your personal keypair will now be on your computer, in your home directory, under the **.ssh** subdirectory. You will need to copy the contents of the public key (**id_rsa.pub**) to a file called **~/.ssh/authorized_keys** on your Raspberry Pi.

If you're in a terminal in your home directory on the computer, you can copy the file over to your Raspberry Pi with this command on GNU/Linux or macOS:

```
ssh-copy-id pi@raspberrypi
```

Replace `pi` with the username you use on your Raspberry Pi, and `raspberrypi` with the host name of your Raspberry Pi.

On Windows, it's perhaps easiest if you open the **.ssh\id_rsa.pub** file in Notepad or another plain text editor and copy its contents. From a terminal program (PowerShell, Windows Terminal, or Command Prompt, use SSH with your password to log into your Raspberry Pi (`ssh pi@raspberrypi`), run `nano ~/.ssh/authorized_keys`, go to the end of the file and paste the key onto a new line with **CTRL+V**, then save the file.

Once that's done, the next time you use SSH to connect to your Raspberry Pi, you'll find that you're asked for your passphrase rather than your Raspberry Pi password. Most operating systems let you avoid entering the passphrase every time you connect. Most systems have some kind of SSH authentication agent. This includes macOS, recent version of Windows, and most modern GNU/Linux distributions.

On Windows, you may need to configure the agent to start automatically. Right-click on the Start menu and choose Windows PowerShell (Administrator), Terminal (Admin), or Command Prompt (Admin), whichever option is available to you. Next, run these two commands and close the PowerShell, Terminal, or Command Prompt:

```
sc.exe config ssh-agent start=auto
sc.exe start ssh-agent
```

With the SSH agent running, the first time you connect to your Raspberry Pi using your key, you'll be prompted for that passphrase, but it won't prompt you on subsequent connections (on Windows, you will first need to run the command `ssh-add` to add the key to the agent). The agent stores credentials securely and unlocks them when you have logged into your computer. Some operating systems, such as Ubuntu, may prompt you to choose whether the key should be automatically unlocked each time you log in.

Once you confirm SSH works without passwords, you can edit **/etc/ssh/sshd_config** to include the line `PasswordAuthentication no`. This will prevent your Raspberry Pi from accepting password-based logins, and will only permit you to login with a key.



LOST KEYS?

The private key half of your key pair (for example, **.ssh/id_rsa**) should be kept secret — but safe, too. Keep a backup of the private key on a memory card in a secure location.

Shared drive

You may be using a service like Dropbox to share files between machines. There is no need to do this on a local network, as the Samba networking protocol on Raspberry Pi lets Windows PCs access it as a shared drive. Samba is already installed in recent versions of Raspberry Pi OS, or you can install it using:

```
sudo apt install samba samba-common-bin
```



SAMBA STEPS

Samba is extremely well-documented, with separate man pages for everything from **smb.conf** to **smbpasswd**, and excellent online books at **samba.org** — look for **smb.conf** examples.

Edit **/etc/samba/smb.conf** with a WORKGROUP value (**Figure 9-2**). For Windows XP and earlier, try **workgroup = WORKGROUP** and/or HOME (For Windows 7 and above). Ensure that Samba knows **pi** is a network user:

```
sudo smbpasswd -a pi
```

Then restart with:

```
sudo service samba restart
```

Raspberry Pi should now show up in Windows Explorer under Network. You can fine-tune **smb.conf** to control what's shared (including printers), and set access permissions.

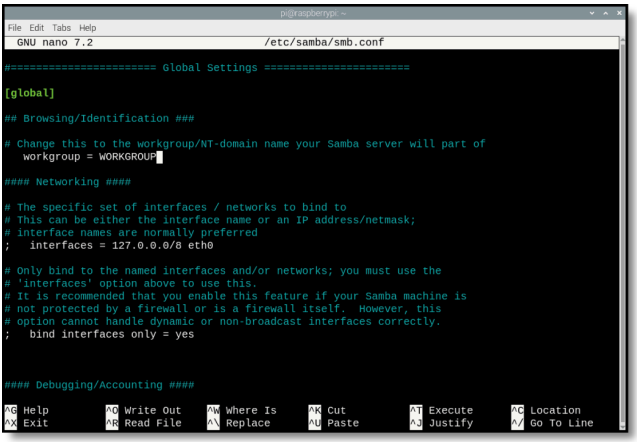
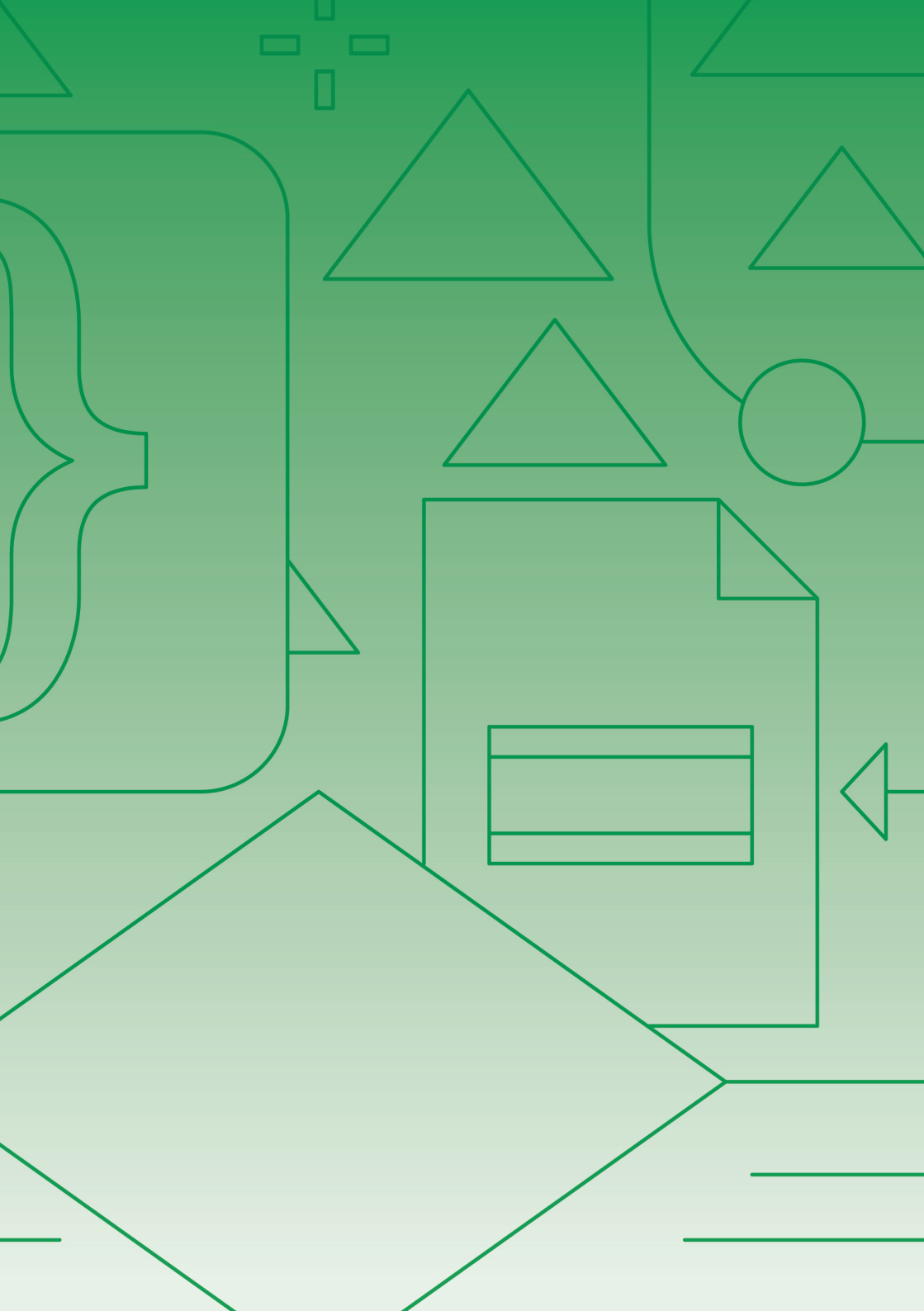


Figure 9-2 There's a lot of configuration in Samba, but simply adding your WORKGROUP name to the default settings should get you up and running



Chapter 10

Downloading and installing

We look at downloading and unpacking software, and show you how to write images to SD cards

Running an `apt` command (see Chapter 3, *Permission to install*) allows access to a huge collection of software — several thousands of packages in the main Raspberry Pi OS repository — but sometimes we need to add software from outside the main repository.

EASTER EGG

Read `man apt`, and you may see: "This APT has Super Cow Powers." If it's there, try typing `apt moo` to see what happens.



If we are lucky, we find that someone has packaged up the software in the **.deb** format used by Raspberry Pi OS, or even created a whole repository to take care of the dependencies. We'll look briefly both at adding repositories, and dealing with other kinds of downloads.

Adding a repository

Information about repositories is kept in the **/etc/apt/sources.list** file, which on a new install just contains the Raspberry Pi OS repository.



Figure 10-1 Downloading and extracting

Rather than editing this file to add other repositories, you are advised to add them in a **.list** file to the **/etc/apt/sources.list.d** directory. To add a new repository, use **sudo nano** to create a **.list** file there (e.g., **extra-sources.list**) and inside it, add a source like this one (combine this all into one line with a space between **main** and **contrib**):

```
deb https://deb.debian.org/debian/ bookworm-backports main
contrib non-free
```

Here, **bookworm** in **bookworm-backports** is the Debian (and Raspberry Pi OS) release name: all Debian releases have been named after characters in the Toy Story series of films since 1996 (former Debian project leader Bruce Perens was involved in the early development of Debian while working at Pixar). Bullseye (Woody's horse) was the name of the previous version.

Most software is in the `main` repository, which can be freely copied or mirrored anywhere. Other components, like `non-free`, allow repositories to contain software you may not be free to pass on, keeping it separate from the main repository.

Once you've added the source, you should run `sudo apt update`. This updates Raspberry Pi OS with the list of the backports packages, a collection of *backported* software from the upcoming version of Debian. Be careful though — they may not work properly. For a full list of what's in that repository, run these commands:

```
cd /var/lib/apt/lists/
for d in *bookworm-backports*binary-arm64_Packages; do
    grep ^Package $d
done
```

This changes to the **/var/lib/apt/lists/** directory and runs a `for` loop over any file matching the wildcard expression. For each matching file, it runs the commands between `do` and `done`, searching each file for lines that begin with 'Package'. You'll notice that the prompt changes from `$` to `>` after you type the line beginning with `for`. The shell is waiting until you complete the `do` command with the `done` keyword.

If you want to see details of a specific file, you can run a command like `apt show -a kicad`. If you have the backports repository enabled, you'll see two versions. Here is the output, truncated to show the version and the repository:

```
Package: kicad
Version: 9.0.1+dfsg-1~bpo12+1
APT-Sources: https://deb.debian.org/debian bookworm-backport...

Package: kicad
Version: 6.0.11+dfsg-1
APT-Sources: http://deb.debian.org/debian bookworm/main...
```

If you run `sudo apt install kicad`, you'll get the older version, because backported packages are always inactive by default. To get the latest package, run `sudo apt install kicad/bookworm-backports`.

Trusting a repository

When you add third-party repository sources to *sources.list.d*, you need to get a key for it and use `gpg` to install it. This is because third-party sources are considered untrusted by default. For example, an apt repository for the R programming language is available by creating a **.list** (such as **/etc/apt/sources.list.d/R.list**) with the following line:

```
deb http://cloud.r-project.org/bin/linux/debian bookworm-cran40/
```

Then try to run `sudo apt update`; you'll get a warning that the repository signature could not be verified. You can add a key with `gpg` to trust the repository — below is the example for the R programming language source.

```
gpg --keyserver keyserver.ubuntu.com \
    --recv-key 95C0FAF38DB3CCAD0C080A7BDC78B2DDEABC47B7
gpg --armor --export 95C0FAF38DB3CCAD0C080A7BDC78B2DDEABC47B7 \
    | sudo tee /etc/apt/trusted.gpg.d/cran_debian_key.asc
```

The specific commands (and key signatures) will be listed with any repository you need to add, so you don't need to remember the alphanumerical code for each one!

wget & curl

Wget is a command to download from a given URL. For example, you can download the Raspberry Pi logo with `wget https://rpimag.co/logo.png`

Wget is a simple but robust download tool, with a powerful recursive feature that helps fetch entire websites, but it does have mild security risks, so be careful using it to fetch scripts. An alternative is curl, a file transfer tool that works with many protocols and can be used for simple downloads. It dumps to stdout by default; to save as a file with the same name as the resource in the URL, use the `-O` switch. For instance:

```
curl -OSL https://rpimag.co/robotfindskitten-euler.tar.gz
```

Here, the `-S` switch will show any errors, while the `-L` switch will enable curl to reattempt to fetch the requested file if the server reports that it has a different location.

Unzip

The game downloaded above is compressed with gzip, which losslessly reduces the size of files, and can be decompressed with `gunzip`. The contents here are files rolled into a *tar archive* (instead of **.tar.gz**, you'll sometimes find they end with **.tgz**), and the `tar` command can do the decompression and untarring in one:

```
tar zxvf robotfindskitten-euler.tar.gz
```

Note that the dash is not needed for single letter options in tar. The first switch, `z`, calls gzip to decompress the archive, then `x` extracts the contents. `v` is verbose, informing you of the process as it happens, and `f` tells tar to work with the named file(s), rather than stdin. Miss out the `z` and tar should automatically detect the necessary compression operation.

The result in this case is a folder containing, among other things, a configure script to prepare the source code for compilation (read the **INSTALL** file first). Before you try these commands, make sure that the C header files for *ncurses* (a programming library for creating text-based GUIs) are installed with `sudo apt install libncurses-dev`

The following will change to the source directory, configure the source code, then compile and install it:

```
cd robotfindskitten-2.7182818.701
./configure
make
sudo make install
```

After you've installed it, you can run the game by running the command `robotfindskitten`. Find kitten, robot!

While gzip is more efficient than zip (and even more efficient options like bzip2 are available), sometimes you'll get a plain old zip file, in which case `unzip` is the command you want.

```
unzip compressedfile.zip
```

Disk image

Disk images for Raspberry Pi OS are available to download manually from **rpimag.co/download,** however you cannot copy it across to a second microSD card (connected to Raspberry Pi via a USB card reader in this case) with regular `cp`, which would simply put a copy of it as a file on the card. We need something to replace the SD card's file system with the file system and contents that exist inside the Raspberry Pi OS disk image, byte-for-byte, and for this we can use a handy little utility called dd.

`dd` converts and copies files — any files, even special devices like **/dev/zero** or **/dev/random** (you can make a file full of zeroes or random noise) — precisely copying blocks of bytes. To copy our image, we will need to unmount the secondary microSD card we've plugged in via a USB card reader. Use `sudo fdisk -l` both before and after plugging in the card (you can also use `df` to see what's mounted) to see attached devices. If, say, a `/dev/sdb` appears, with the size equal to the SD card, then unmount with `umount /dev/sdb1`. Now copy the disk image with:

```
sudo dd if=~/Downloads/2024-11-19-raspios-bookworm-armhf.img \
        of=/dev/sdb bs=1M
```

Development of Raspberry Pi OS's ancestor UNIX started in 1969, so we've covered a few utilities with a long heritage in this book, but that `if=` in place of the usual dashes for command-line options indicates a lineage stretching back to the early 1960s, and IBM's Data Definition (DD) statement from the OS/360 Job Control Language (JCL).

Be very careful that the destination matches the correct disk, or you will lose the contents of another storage device! The `bs=1M` is a block size default; `4M` would be another safe option. Now put the card in another Raspberry Pi and go and have fun!



Chapter 11

Start and Stop at your Command

We take a look at scripts to manage the way Raspberry Pi OS starts and shuts down

Raspberry Pi OS, much like every other GNU/Linux distribution, uses a very specific software suite (*systemd*) on startup that makes sure all the relevant software you need is started and loaded before you log in.

The first process the kernel starts is **/lib/systemd/systemd** (with a PID of 1), and it is the parent process of everything that happens in user space once the Linux kernel has finished initialising devices and drivers, and mounted the file system. If you run **`ps auwx`**, you'll see **`init`** listed as PID 1, but take a look under the hood with **`ls -l /sbin/init`** and you'll see that **`init`** is a symbolic link (see "Linked ln" on page 80) to **`systemd`**:

```
$ ls -hgo /sbin/init
lrwxrwxrwx 1 20 Mar  6 09:56 /sbin/init -> /lib/systemd/systemd
```

So, **`systemd`**, masquerading as **`init`**, gets everything else started, and usually ends with the prompt inviting you to log in to your Raspberry Pi. Raspberry Pi OS hides most of the messages that this startup process generates, but you can see them by typing **`dmesg`**.

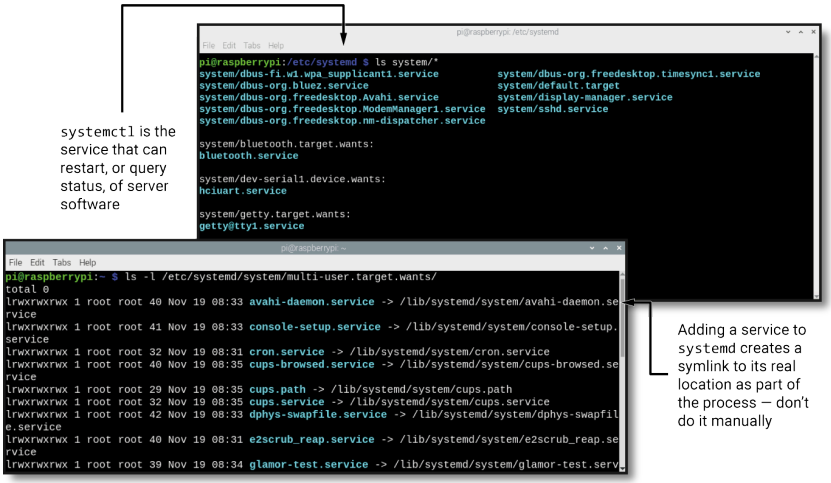


Figure 11-1 Raspberry Pi OS service management

Systemd

systemd starts services in parallel, and can defer service starts until they are needed. Rather than many scripts for individual components, a target is set, and **systemd** resolves the dependencies until it reaches that target, avoiding any fixed startup sequence along the way.

Files are found under **/etc/systemd/system** and there's a lot to learn. The one thing to remember for any user is that **systemd** and its service manager are controlled with the **systemctl** command.

```
sudo systemctl restart ssh
```

...will restart the SSH server — something you'll need to do if you edit **/etc/ssh/sshd_config** to change the port it listens on, for example. For backwards compatibility, as well as **/etc/init.d** scripts to start and stop services, the system of service commands that worked on older versions of Raspberry Pi OS, such as:

```
sudo service apache2 reload
```

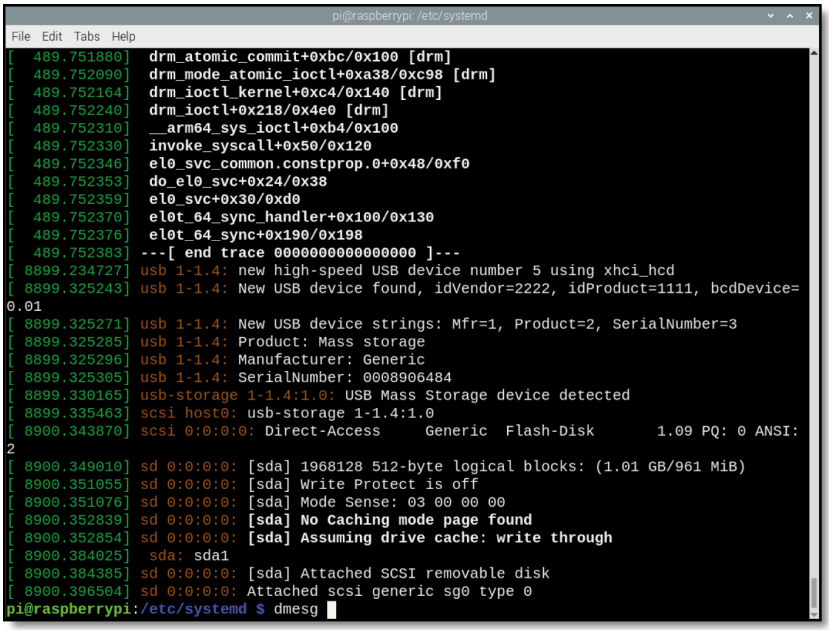


Figure 11-2 `dmesg` gives you the startup messages from the kernel — nuggets of useful info buried in plenty of legacy boot information

...still works (here we ask a running Apache2 Web server to reread its configuration files).

Under **/etc/systemd/system/multi-user.target.wants** you will find files like **cron.service** which, when examined closely with `ls -l`, you'll see are links to files of the same name in **/lib/systemd/system** (other distributions may place the files under **/usr/lib**).

SYSTEMD DOT

Systemd is a complex system, with many elements — from **systemd.unit** to **systemd.slice** — but they all have their own man pages, and it's worth reading through to help get the overall concept.

Don't worry if what's inside these files looks confusing; there's a logic to them with their conditional dependencies, but you can safely forget about them until you need to get some software working automatically on every system restart for your Raspberry Pi project. Even then, we'll show you another way with crontab — otherwise you'll need to be aware of the following, as those links aren't created manually:

```
sudo systemctl enable postgresql.service
```

...will create the link, enabling PostgreSQL to run on startup (assuming it's installed). To make `systemd` aware of configuration changes (such as when editing **/etc/fstab**) before the next restart:

```
sudo systemctl daemon-reload
```

Linked ln

`systemd` makes links between files automatically, but there will be times you'll want a file to appear to be in a local directory when it is elsewhere, with a handy little command we have not so far had a chance to show you: `ln`.

ln makes a link which allows a file to effectively exist in two places at once. In the following example:

```
ln -s /usr/share/code-the-classics/README.md README-ctc.txt
```

...a file will appear in your current working directory. But `ls -l` and you'll see that it's a special type of file, a symbolic link pointing to the actual file. Edit **README-ctc.txt** and you'll find that **README.md** in the linked directory will be edited.

Soft, or 'symbolic' links, are created with the `-s` switch — you don't even need the file to which you're linking to actually exist, which makes it handy if you're linking across a network, or to a removable drive.

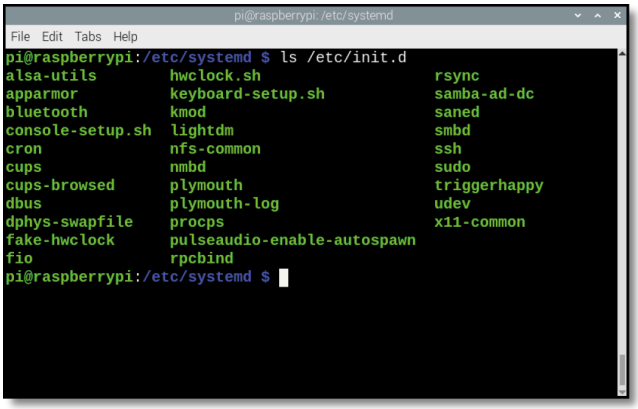


Figure 11-3 Backwards compatibility with much older init.d scripts — and even run levels — is maintained by systemd

It's called a symbolic link because it works by linking to the name of the target file, rather than to the file data itself. Create a new file and a hard link:

```
ls > listing.txt
ln listing.txt listing2.txt
```

...and you have two names (and locations) for the same file — sounds like the same thing? Not exactly: if you delete the original file in the first example, you can replace it with a new file of the same name, but different contents, and the symbolic link will point to the new file. Remove the original file in the hard link case, and the link still points to the original data.

Location, location

The startup scripts are generally for daemons, processes you want running all of the time, like web servers and databases. There are plenty of programs which do housekeeping that need to be run periodically — hourly, daily, weekly. For this purpose, the cron software utility is ideal

for scheduling the running of such programs and tasks. Cron searches its configuration directories and runs the scripts it finds there — have a look at the different folders in **/etc** with names beginning with 'cron'.

The easiest way to get to know where things like this are on your system is to search with **`locate`**. First, enable the database update service with **`sudo systemctl enable plocate-updatedb.timer`**, which will update locate's database daily. You can update it immediately with **`sudo updatedb`**, then try something like **`locate cron`** — which will find you every file or directory with cron as its name or as part of its name.

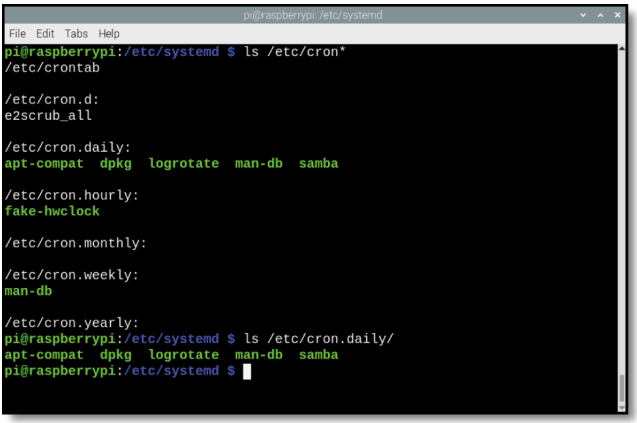


Figure 11-4 Examining some of the cron directories

The built-in alternative is **`find`**, a powerful utility which enables you to search particular directories — or the whole file system — by name or name fragment, size of file, how long ago they were modified, or whether they're bigger than another file — enough to deserve a whole chapter of its own. Because it searches the file system, rather than a cached listing, it takes longer than locate, but it is always up to date, and has search options not found in locate (see "FIND AND REGULAR EXPRESSIONS" on page 83). To replicate our **`locate cron`** command with the **`find`** tool:

```
find / -name '*cron*'
```

FIND AND REGULAR EXPRESSIONS



The `find` tool can search by regular expressions, as well as (part) name. They're a whole book topic in themselves, but well worth investigating once you've got command-line basics under your belt, as regexps can be used with many commands.

If you were looking for cron or crontab, but not anacron, you could search for 'cron*' instead. There will be more output than you want, so pipe it through a pager, or perhaps a grep. Back to using cron — the easiest way is via crontab, which maintains a table where each row specifies a command, and how often it is to run.

You edit the crontab file not directly, but with `crontab -e`, which calls up the default editor to do it. Here's an example that helps keep track of how much free disk space you have on your primary disk:

```
45 05 * * 1-5 df -h / >> diskspace.txt
```

... which grabs the output from `df -h` and appends it to the file *diskspace.txt* in your home directory every weekday at 5:45 AM. The first five crontab fields cover minute, hour, day of month, month, day of the week, and can all be replaced with a single special value, like `@daily` or `@hourly`. While `man crontab` tells you a little about crontab, `man 5 crontab` is far more useful as it covers the layout of the file, with examples. Run `man man` for more on the numbered sections available with `man` commands.

A fresh startup

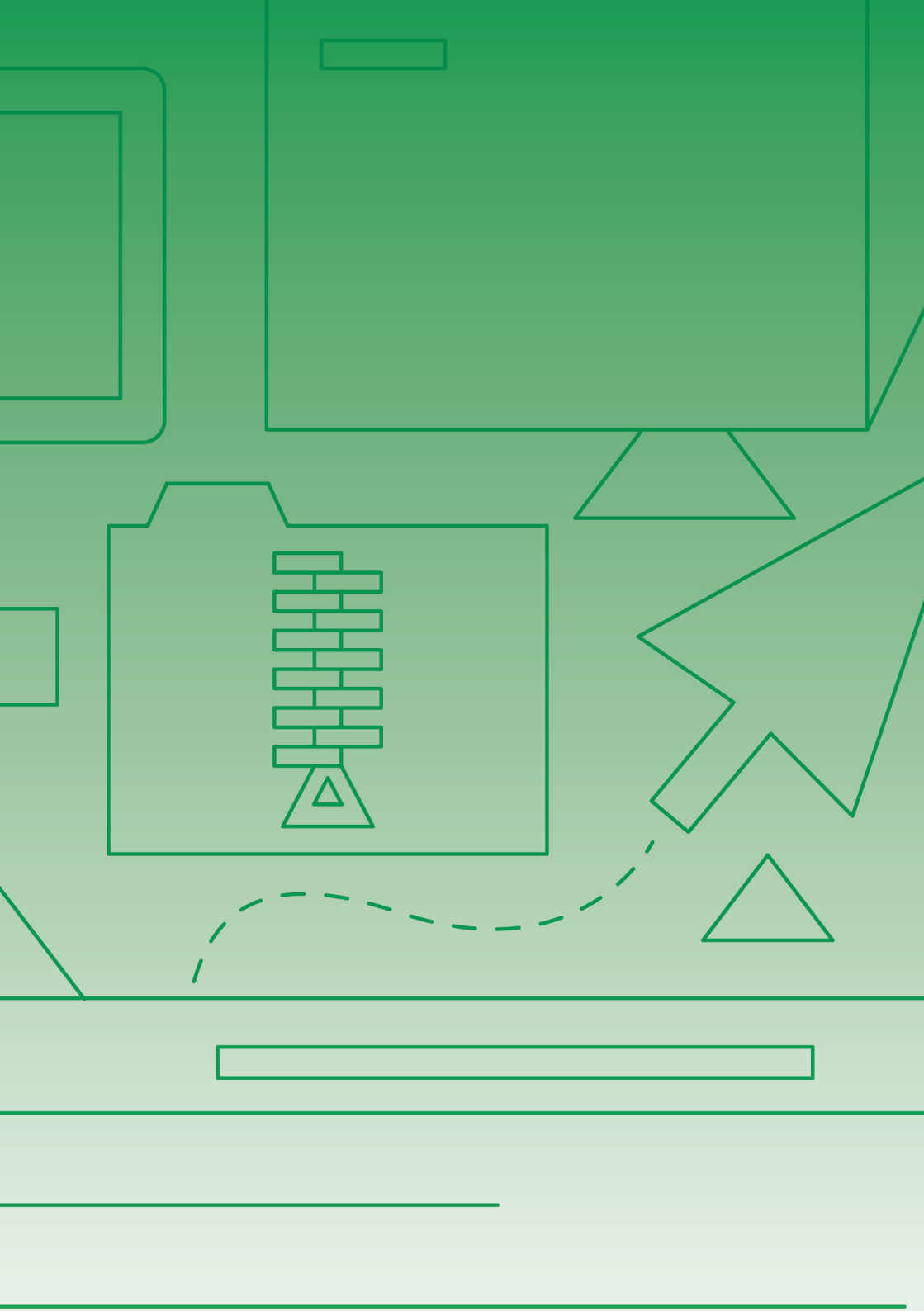
Using the value `@reboot`, we can easily run our own scripts on startup, without messing about with system startup scripts. There are times when a full `systemd` startup script will be more appropriate, but for quickly getting something tested, put the script into crontab.

There are two things that may catch you out. Firstly, you might be running scripts out of a directory that you have in your `$PATH`, which defines where Bash looks for commands. As `$PATH` is only set once you log in and your personalised **.bashrc** file is read, scripts running from crontab which

are run immediately upon startup will not be aware of your `$PATH` setting. So, you will need to express some commands by their full paths, such as **/home/pi/bin/test.sh** — but in general, cron should be able to find commands in **/bin** and **/usr/bin**. You can replace the line you added previously with one that runs the command each time you reboot:

```
@reboot sleep 10; df -h / >> diskspace.txt
```

`systemd`'s parallel service startups also mean that some services, such as the network, may not be ready when your `@reboot` commands in the crontab entry are called, but you could use the smallest time delay that consistently works on testing. You can also use the automation of cron to run your own backup scripts, so we'll discuss backup options in the next chapter.



Chapter 12

Save it now!

Learn how to protect your data with backups and disk wipes

Perhaps you've got backups running automatically on your main computer, or perhaps you just back up what's important now and then — it's OK, we're not here to judge. But we will say that anything you don't have backed up, you don't have. Computers break. Hard disks and SSDs break. Accidents happen. The unexpected is somewhat inevitable.

IT professionals prepare as if they could lose everything at any moment, at least the best do. That might sound a tall order for a small single-board computer you bought at a low price to use in a hobby project, but your data is the most important thing on it, and good backups are a useful discipline to take elsewhere. Fortunately, the command line can actually make this easier; we'll show you some of your best options for (relatively) painless backups.

Simplest of all is to copy the data and move it elsewhere. It's labour-intensive, compared to automatic solutions, but for only-very-occasional backups it's certainly better than nothing, so we will not ignore this option.

Whether you're copying to disk, or moving the backup to another machine, it's best to make it as small as practically possible. The zip compression format may not compress as much as some specialist UNIX choices, but it

does mean that other users should be able to open the file(s) with their standard compression software. You could go with LZMA or bzip2 compression (from `tar`, using `-J` or `-j`, respectively) for better compression to a smaller file size, but although alternatives to gzip save a little more space, they can take far longer to perform the compression.

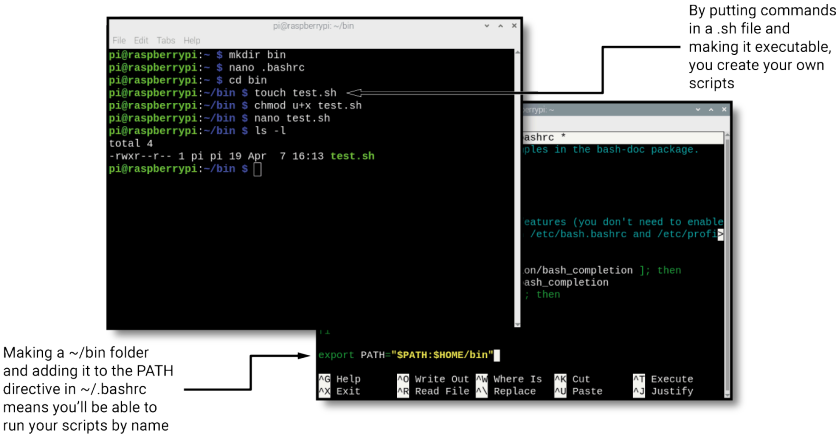


Figure 12-1 Making and running scripts

You can use a single `tar` command to gather all the files in one directory and compress it into an archive:

```
tar czvf mybackup.tgz myfolder
```

The `c` switch tells tar to create the archive, diving into all subfolders found; `f`, use the named file (here, **mybackup.tgz**), is necessary to direct the output away from the terminal, or — historically — a tape device. Yes, tape, because `tar` is short for *tape archive*, as telling a sign of its longevity as its frequent use without a dash in front of the command-line options. `v`, as with many commands, asks for more verbose output, so the program tells you what it is doing and what (if anything) has gone wrong.

Lastly, `z` invokes gzip compression — saving the extra step of creating a .tar archive, then running it through gzip. To unpack the archive, substitute `x` for extract in place of `c` — you can omit the `z`, as tar will recognise the compression type and automagically deal with it (this will overwrite

the files you just archived if you run this command from the same folder where you created the archive):

```
tar xvf mybackup.tgz
```

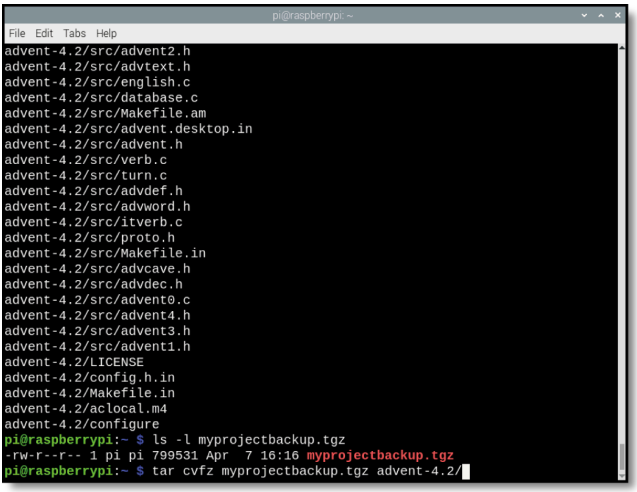


Figure 12-2 A single command wraps up all of the files and subfolders, then compresses the tar archive with gzip

TAPE ARCHIVE

tar dates from the days when computers backed up to big tape reels, those essential props of 1960s and 1970s sci-fi films. The lack of a file system on tapes means that **tar** must save all of the file system info such as ownership and timestamps into its own format.

A safe home

Often, using tar to back up your personal files — with **cd /home**, then **tar** on your home folder — will be all you need, but if you have data across directories, from **/etc** to **/var/www**, it's simplest to back up the entire microSD card. We looked at copying a new Raspberry Pi OS image onto a card in Chapter 10, *Downloading and installing*; backing up your disk is

almost a mirror image of that process, which you can easily do on another GNU/Linux computer with a USB card reader. Power down your Raspberry Pi, put the SD card in the card reader, and connect it to the other computer. If your computer automatically mounts it, use `sudo umount` to unmount it.

You'll be creating a file as big as the entire SD card — usually 16GB or more — so make sure the computer you run this command on has enough space. It helps that you will be compressing the image file as it is created, which, for a Raspberry Pi with a modest amount of data on its SD card, will result in an image of 4GB or more. Look back at "Disk image" on page 74 for how to be sure which device the USB card reader is — for a card plugged in as **/dev/sdb**, and unmounted, do:

```
dd bs=4M if=/dev/sdb | gzip > rpiback.img.gz
```

Open another Terminal tab and monitor your disappearing disk space with `df` — if you don't think that it will fit, stop the `dd` operation with the usual **CTRL**+*C*, then `rm` the image file that you have partially created, and go and perform the backup on a computer with more disk space — or with a backup drive mounted, to which you copy the archive directly. Turning the backup into a usable microSD for the Raspberry Pi means piping the other way, from `gzip` to `dd`:

```
gzip -cd rpiback.img.gz | dd bs=4M of=/dev/sdb
```

For disk operations like dd, you'll need root permissions: you can prefix `dd` with `sudo`, but for saving the file outside of your home directory, you may also need `sudo` — which means typing it in front of `gzip` as well.

This is only mildly inconvenient on Raspberry Pi, where `sudo` does not demand your password — but on other computers, you need a reliable way of becoming the root user for every operation: running `sudo -s` will give you a shell with root permissions, but remember to `exit` afterwards. Alternatively, a chain of commands can be run with full admin permissions. For example, to restore your backup from a file to an SD card:

```
sudo bash -c "gzip -cd rpiback.img.gz | dd bs=4M of=/dev/sdb"
```

Remote copy

It's good to be able to make backups as required, using removable drives, but to move towards systematic backups you will need to copy across the network. We mentioned SCP in Chapter 9, *Remote Raspberry Pi*. To copy your backup file to another machine, one that is running the SSH server, pass your login name with the command:

```
scp -p rpiback.img.gz pi@192.168.0.207:/home/pi/bak/
```

You will then be prompted for the user password. Change the **pi@** to whatever your username is on the remote machine, not the Raspberry Pi you're copying from. The **-p** preserves information such as when the files were last accessed. Note that **-P** (capital p) can be used to specify a particular port number. Make sure you create the destination directory with **mkdir** before you copy the file.

> **WHICH FILES?**
>
> Apart from ~, your project may have config files in **/etc** or even **/opt**, and under **/var** are logs, web config and files — all of which you may have modified for a project.



We showed you how to set up a Raspberry Pi with a fixed IP address in Chapter 7, *Predictable Networking*; that Raspberry Pi, with a plugged in USB disk drive, could be an inexpensive backup machine, as well as media server or whatever else your home or office needs.

Because you're sending these commands through the Bash shell, you get all the usual Bash advantages, from tab completion (just type **bac**, or however much of the file name is unique in your present working directory) to wildcards. If you have disparate archives in the same directory — such as **www-backup-20241225.gz** and **data-backup-20241226.gz** — then copy them all with:

```
scp -p ./*backup*gz pi@192.168.0.207:/home/pi/bak/
```

So far, so good, but if you want to automate your backup process, the interactive element — typing a password — would be better avoided. As long as you can maintain security in some other way, of course.

Efficient backups

Back in Chapter 9, *Remote Raspberry Pi* when we set up our SSH server, we generated keys with `ssh-keygen` — these keys can be used to provide passwordless login. You can copy them across to other machines as we did in "A secure key" on page 64, and you'll be all set to automate remote backups — but if you create a tar file every time, you'll waste a lot of disk space duplicating data that changes infrequently.



PASSWORD FREE

Using your key to log in is a convenience you quickly get used to — beyond `scp` to your backup server, try it on any machine under your control that you have to log in to.

`rsync` lets you copy data in much the same way as `scp`, but uses a *delta-transfer algorithm*, to only transfer the difference between the copies of the source file on your disk, and the remote, saved version. This both saves bandwidth used and avoids cluttering up your backup disk with multiple near-identical versions of a file. It's also handy if you're paying a cloud provider for storage and data transfer.

If your version of Raspberry Pi OS doesn't have rsync, it's just an `apt install` away. Typically, rsync uses SSH for transport, but you can set up a server running an rsync daemon, and directly contact the *rsync://* URL over TCP (defaults to port 873). In this case, set an `RSYNC_PASSWORD` environment variable or use the `--password-file` switch.

`rsync` is just one of many tools for archiving your precious data. In addition, there is the possibility of using version control systems, such as git, for both backing up, and tracking changes on, important files.



WHY REMOTE?

As well as being able to centralise backups for more than one machine, a remote backup protects you from unexpected disasters such as fire or flood where your Raspberry Pi is. Fairly unlikely? Yes, but that doesn't stop you insuring your house.

Script-it-yourself!

But let's row back to simpler commands. We have seen from early on how powerful Bash can be by chaining together a few commands; another way of putting commands together is to bundle them into a script — a short program simply comprising a small number of Bash commands and known as a shell script. Take a look at this code — try typing it in to your favourite text editor, adjusting it for the IP address of your networked backup server, and backup folder location (or change the `scp` line to a `cp` to a plugged-in backup drive), and saving it as **test.sh**.

```
#!/bin/sh
cd /home/pi
tar czf mydocsbackup.tgz Documents
scp mydocsbackup.tgz pi@192.168.0.207:/home/pi/bak/
```

HASH BANG



The shebang, or hash bang — `#!` — at the start of the script is an instruction to the program loader to run the program immediately afterwards — in this case an interpreter directive to run **/bin/sh** — and pass the script as an argument to it.

Then make the script executable with:

```
chmod u+x test.sh
```

...and run it with `./test.sh` — any problems, then check the names, network address, and did you copy your key over to the other computer?

Now we have a script that saves a folder, and copies it remotely, do you notice any potential problems? Each time you run it, it will overwrite the previous **mydocsbackup.tgz**, both locally and remotely. We need a way to put a timestamp on the backup name:

```
#!/bin/sh
TODAY=$(date +"%F")
cd /home/pi
tar czf mydocsbackup-"$TODAY".tgz Documents
scp mydocsbackup-"$TODAY".tgz pi@192.168.0.207:/home/pi/bak/
```

What we have done is set a variable — `TODAY` — to the current date, in YYYY-MM-DD format, which we can now access with `$TODAY` (remember, we permanently set variables for the shell named this way, when changing the prompt in Chapter 5, *Customise the command line*). You can run `date +%F` in the terminal — `date --help` will show you the many other format options. Now you can automate it by putting the script somewhere like **/usr/bin** (and with a better name than **test.sh**), and running it regularly with cron, as we covered in Chapter 11, *Start and Stop at your Command*.

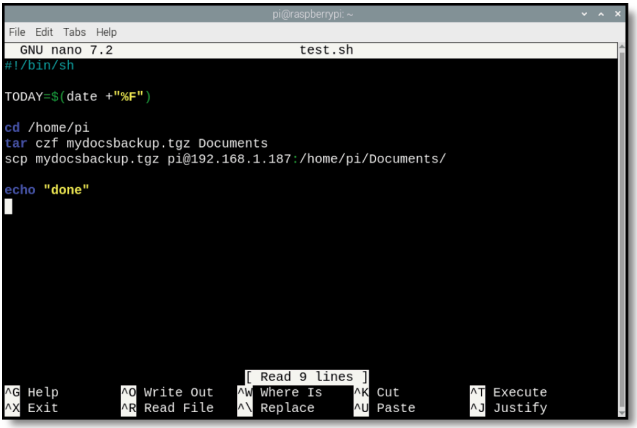


Figure 12-3 A datestamp in our script means that we are not producing a backup with the same name each day we run it

Shell scripts tend to grow; there is always room for improvement. Here, you may want to back up more than one directory, for example, or use `echo` to let users know what the script is doing at each stage. You could even make it interactive, letting users choose which directories to back up.

There are plenty of shell scripting tutorials online, and great books to take it further, but Raspberry Pi OS itself holds many great shell scripts, from which you can learn by example. To see how a script can be organised to still be maintainable with over a thousand lines of code, have a look at **/usr/bin/raspi-config**.

However grand or modest your scripting ambitions, don't be afraid to try things out: build up gradually, and test your code each time, so that you know where to look for any errors that you introduce. Help is at hand

from the Raspberry Pi forums, and pasting your code into **shellcheck.net** will give you valuable feedback — for example, advising you the `cd` line of our backup script should be:

```
cd /home/pi || exit
```

...in case the `cd` step fails — this is generally a good idea, although not so important in this particular case. Now that we have a choice of backup options, one task remains: securely getting rid of data from disks. This is a concern for anyone handling other people's data, or just protecting their own privacy and security.

Through the shredder

Back in the 1960s, if you wished to cover your tracks, Mission Impossible told us it was done by the show opening's taped mission assignment finishing, "This message will self-destruct in ten seconds...", and boom went the tape player.

In these digital days, protecting privacy or security means understanding how a disk drive actually stores data, so that you don't dispose of old disks under the mistaken impression that you've securely erased data, when you haven't.

Disk space is collected into blocks, sized typically at 4096kB, that are indexed by the file system with *inodes* so that the disk controller knows where to send the read head to retrieve information. SSDs and flash drives don't have read heads, but still organise the data in a similar fashion. Most disk operations occur at the inode level — so moving a file between directories on the same disk partition is simply done by relabelling an inode. `rm` does not delete the data stored, just the reference to its blocks on the inode.

Plug in a disk drive after someone has done `rm -rf` on it and it will look empty, but use a low-level utility and you'll have access to all of the jigsaw puzzle pieces needed to put the data back together again. So far, so NCIS, but can this be significant to the average Raspberry Pi user? Given the range of projects out there, and the multifaceted data that they collect,

to stay on the right side of new and future data protection laws it will be useful to know how to securely remove data from your disks.

Caution

Before we start erasing disks, think of the carpentry maxim, 'measure twice, cut once'. It's easy to mistakenly erase the wrong disk or partition if you're not paying attention. Given enough opportunities, most of us do it, and it's often the lesson that teaches us to make proper backups! Running through other operations on the disk (`mount`, `df`, `ls`, `umount`), before erasing, works as a sanity check that you're addressing the correct partition.

Now to those blocks. You can overwrite every bit of information, either with all 1 digits, or with totally random data, using `dd`. Even then, with magnetic disks, it's theoretically possible to recover the data, and multiple overwrites will be necessary — but before you worry about writing a script to do that, let us introduce `shred`, a utility that does just that, overwriting with as many passes as you select as a command switch (or defaulting to three):

```
shred -vf -n 5 /dev/sdb
```

Adding a `-z` switch will overwrite the random data shred has used with zeroes, leaving a new-looking disk. `-u` will delete the file after the secure overwrite. Shred can also safely overwrite and/or remove individual files.



WHERE AM I?

If you follow our tip on SSH keys everywhere, and end up hopping from machine to machine, remember to customise your Bash prompt (see Chapter 5, *Customise the command line*) so that you are always sure on which machine you're about to erase a file.



Chapter 13

Easy compilation

Here we look at how to install software and build it from the source code

It's so easy to install most software on Raspberry Pi OS — provided it's mature enough to have been packaged as a **.deb** archive. Often, however, there's some great code available you'd love to try, but it asks you to compile it — sometimes after first cloning it from GitHub.

While not so straightforward as running an `apt install` command, there's little to fear in stepping through the decades-old ritual of compiling software — and when errors do occur, it's often quite easy to get back on track. We'll also look at scripted installs and Python packages, but first let's find out what to do if a package is in the wrong format.

Raspberry Pi OS is just one of many distributions based upon Debian — **distrowatch.com** lists more than a hundred — and the ones you're most likely to have heard of include Ubuntu and Linux Mint. Ubuntu — the name derived from an Nguni Bantu term meaning 'humanity towards others' — is so popular that many projects, including ones not in the Debian and Raspberry Pi OS repositories, maintain **.deb** packages that you can download and install.

Ubuntu also introduced the idea of Personal Package Archives (PPA) to the Debian family — special software repositories for uploading source

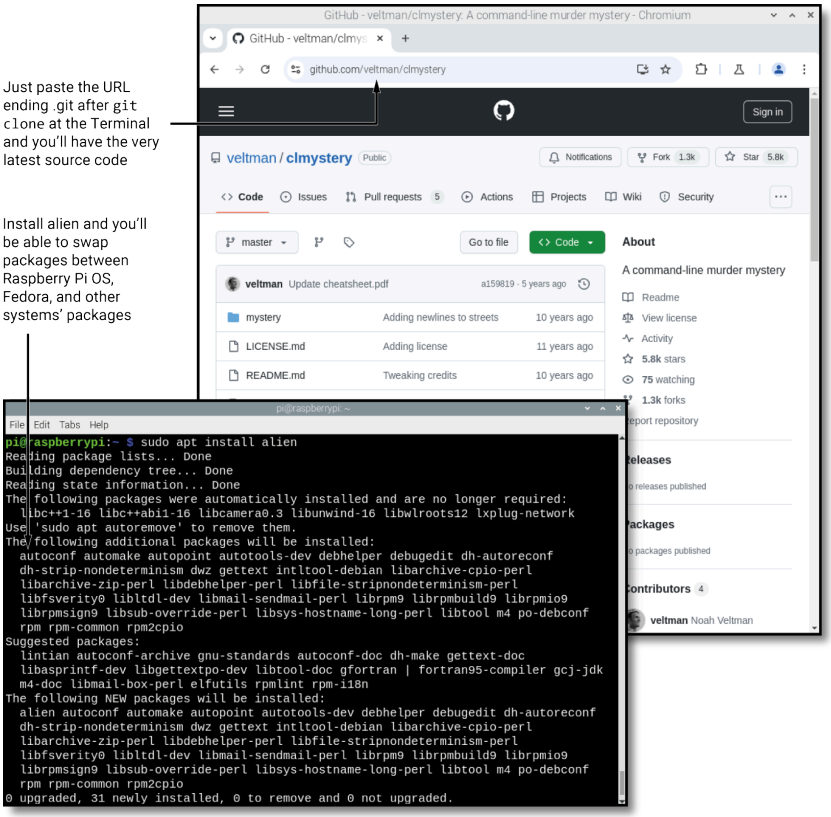


Figure 13-1 Source code and alien packages

packages to be built and published as an APT repository by Ubuntu's Launchpad software like the official Ubuntu repository, but for unofficial software from outside. You won't tend to find these with Raspberry Pi software, but if you've been inspired to try one of the relatives of Raspberry Pi OS on your main PC, you'll find plenty of instructions on them in the Ubuntu community documentation.

When you run **`apt`**, or you **`apt search`** to look for a package, APT quizzes its local record of what packages are available to it. The record of where it gets these packages from — the address of the repositories — is kept

in the file **/etc/apt/sources.list** and at files in **/etc/apt/sources.list.d/** to which you can also add repositories by hand should a project you are interested in offer one.

You can also edit **sources.list** (not advised), and meanwhile, plain old **.deb** files can be downloaded, and then installed with the command:

```
sudo dpkg -i example.deb
```

And any missing dependencies resolved with an:

```
sudo apt install -f
```

There is another popular family of Linux distributions, based upon Red Hat, and including Fedora and CentOS. Fedora is available as an alternative to Raspberry Pi OS should you wish to try it out. What we're concerned with here is situations where you may need to install packages for one distribution, onto a system of the other type. Your friend here is Raspberry Pi OS's `alien` package, which will convert software between packages in **.deb** and **.rpm** (RPM Package Manager, yet another recursive acronym) format (you'll need to install it with `sudo apt install alien`).

To convert from **.rpm** to **.deb**, use `sudo alien some-package.rpm`. You can go the other way with `sudo alien -r mysoftware.deb`.

Compiling code

Before package managers, it was normal to compile your own software, obtaining source code — usually written in the C or C++ languages — and running it through the GCC compiler, before linking libraries, and installing to the correct place on your disk drive.

Many of the headaches involved in the process are long gone as `configure` and `cmake` scripts do all of the hard work, checking dependencies are installed, then running the correct compiler flags for the project and the platform, and even installing the man page in the correct place.



TRACK YOUR INSTALLS

Installing software from outside of Raspberry Pi OS's repository means it must be looked after separately: updates, bug-fixes, security patches, and disaster recovery if something happens to your Raspberry Pi. Keep track of what you install.

Although most of the software you'll want to run will be available as a **.deb** to install with **`apt`** or **`dpkg`**, or come with a shell script which deals with installation (see below), plenty of projects, particularly those you'll find on GitHub or Bitbucket, need unpacking then compiling.

After unpacking the archive with **`tar xvf latest-software.tgz`**, **`cd`** into the directory created, and look for a file with a name like called **INSTALL** or **README**. It will usually tell you to run a common sequence of commands — but read the instructions as there are variations. Two common patterns you'll encounter are:

```
./configure
make
sudo make install
```

and:

```
mkdir build
cd build
cmake ..
make
sudo make install
```

In most cases, software you build in this way will be installed somewhere in **/usr/local**, with executable files stored in **/usr/local/bin**. This is done to avoid interfering with versions of software that are installed through the package manager.

Dependencies

Commercial software (open-source or proprietary) often comes as a large binary file containing all of the dependencies, statically linked into the

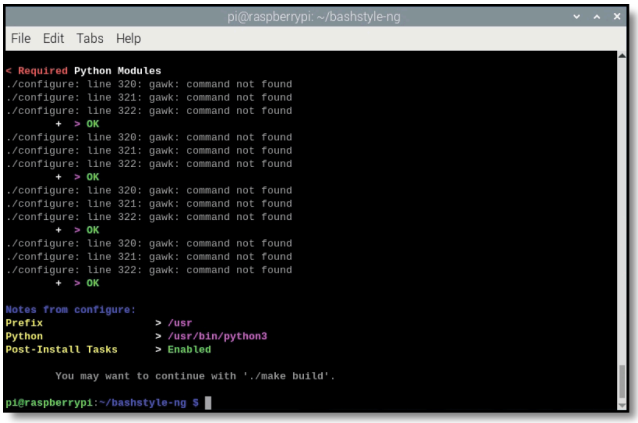


Figure 13-2 Compilation looks complicated, but most projects provide a well-crafted makefile for the process

application. With most non-commercial software projects, however, it is more common to just get a list of which version of which library will be needed to compile and run the software. Luckily, for a standard Debian package, there's a simple command that will install all the dependencies required to build it:

```
sudo apt build-dep <package name>
```

However, occasionally you'll need to compile and install something else first. Often the reason for compiling is to get the very latest version of the software, for new features, compatibility, or bug fixes. In the last few years, GitHub has become the default place to host free software projects; other repositories are available, but we'll just look at fetching software from GitHub, to quickly show you what you need to know.

Your first encounter with GitHub's existence may be seeing a banner on a project page inviting you to 'fork me on GitHub', or an invitation to 'clone' the software. Yes, we're in the world of a project big enough to sustain its own jargon. A fork is simply your own development copy to work on, after which you can offer the changes back to the project, or publish them (on GitHub or elsewhere) for others to try or build upon.

You don't even need to be able to code — many people are now using GitHub for collaborative development of documentation, including scientific research, and even fiction. But to simply download something, we don't need to worry about other Git methods — just 'clone' the application's source to your Raspberry Pi:

```
git clone https://github.com/veltman/clmystery.git
```

...which will make a local directory containing all of the source files. `cd` into the subdirectory just created. This is The Command Line Mystery — a text game which teaches you command line use. Entertaining *and* useful.

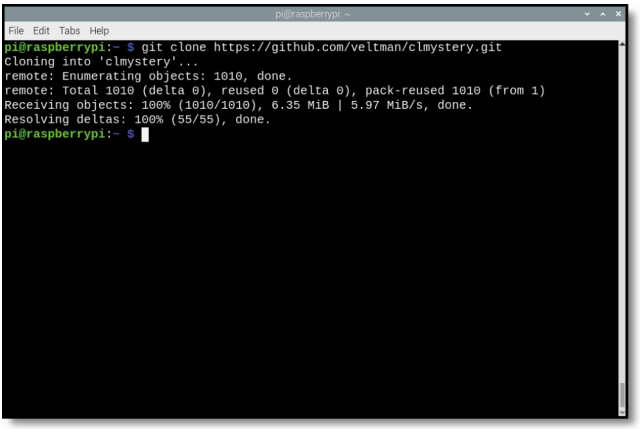


Figure 13-3 Git was developed by Linux creator Linus Torvalds to handle the complexities of multi-million lines of kernel code — yet makes it simple to maintain version control on the simplest of software

Nil desperandum

Sometimes, somewhere along the compilation, something goes wrong, and the script terminates with a complaint of some missing package. On a good day, you'll `apt search` for the name of the missing dependency and there it will be, easy to install on Raspberry Pi OS.

On a slightly-less-good day, you're going to have to dig around a bit to find the software, or the latest version that's being called for. Maybe hav-

ing to go to GitHub or *SourceForge.net*. This is OK if the developer has a setup not too far removed from your own; the extra install steps may go smoothly. Otherwise, don't despair: help is available.

Most projects have one or more ways of reporting a problem and seeking help: a mailing list or Discord server; a wiki on the project SourceForge page; an email address for the lead developer; or even a social media account. Try to state your problem with plenty of detail, and remain polite and patient, and usually you'll find helpful people. Remember, although people want to be helpful, you are asking them to give up time to answer you — if you're frustrated with the installation process, and we've all been there, don't let that stop you being respectful of anyone going out of their way to help you.

If you don't have time for chasing up answers, but remain interested in a project, another option is to wait for the project's next release. By this time the dependencies may have become more widely available, even making their way into the Raspberry Pi OS repositories, or the problem may simply no longer arise.

A scripted install

You'll sometimes be asked to grab an installation script and run it directly — as is the case with the excellent EduBlocks, a half-way house between Scratch and Python which is helping young coders get over the large step between the two languages. The EduBlocks install wants you to run:

```
curl -sSL get.edublocks.org | bash
```

...which takes the shell script at **get.edublocks.org** (you can look at it there in your browser) and pipes it through to a Bash process to run. The `-s` switch tells curl not to show a progress meter or error messages; `-S` overrides this to show an error message if it fails; while `-L` tells it to follow to wherever the site redirects to for the script.

If you don't like running software without knowing what it is doing to your Raspberry Pi, or simply wish to take a look inside the script and see what the installation does, instead, download and save it as **edu-install.sh** or

just **install.sh** if that won't overwrite anything of the same name in your current working directory:

```
curl -o install.sh -L get.edublocks.org
```

...and you'll see the script downloads and unpacks a tarball with more than one script inside, first running the one that installs dependencies.

You can run the downloaded script with `sh edu-install.sh`. If you don't read the script before running it, you're placing a lot of trust not just in the developer(s) who wrote it, but every step of its journey before it reached your machine.

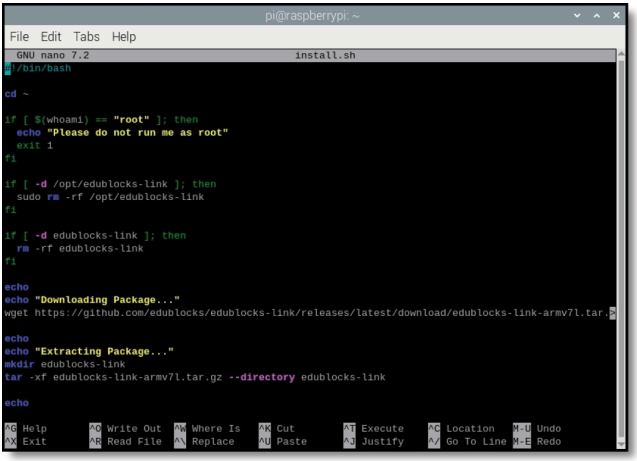


Figure 13-4 EduBlocks is a good example of a project that makes your life easy by putting every stage of the installation into a shell script, and giving you just one command to run the installation

Python, please

Over the years of installing software from various sources, you learn to recognise signs of whether or not it's likely to be a painless installation — and one cause for hope is something written in Python. Although both good and bad software can be written in any language, Python software and Python libraries just seem to be well-packaged and reliable.

Although Debian-based GNU/Linux distributions like Raspberry Pi OS come with the excellent APT and dpkg (Debian Package Manager), many popular programming languages have evolved their own ecosystems of packing tools and repositories — there are several for the Emacs text editor alone! Two that you are most likely to come across are JavaScript's `npm` and Python's `pip`.

Starting with the version of Raspberry Pi OS based on Debian Bookworm, you need to set up a Python *virtual environment* before installing a Python library with `pip`. This lets you install optional libraries without affecting your main Python environment. The simplest thing to do is set up an environment under your user account. Use the following command to create a virtual environment in the **env** folder in your home directory:

```
python -m venv --system-site-packages ~/env
```

Next, you can run the following command from any directory to start using the virtual environment (you'll need to run this for each new command line session):

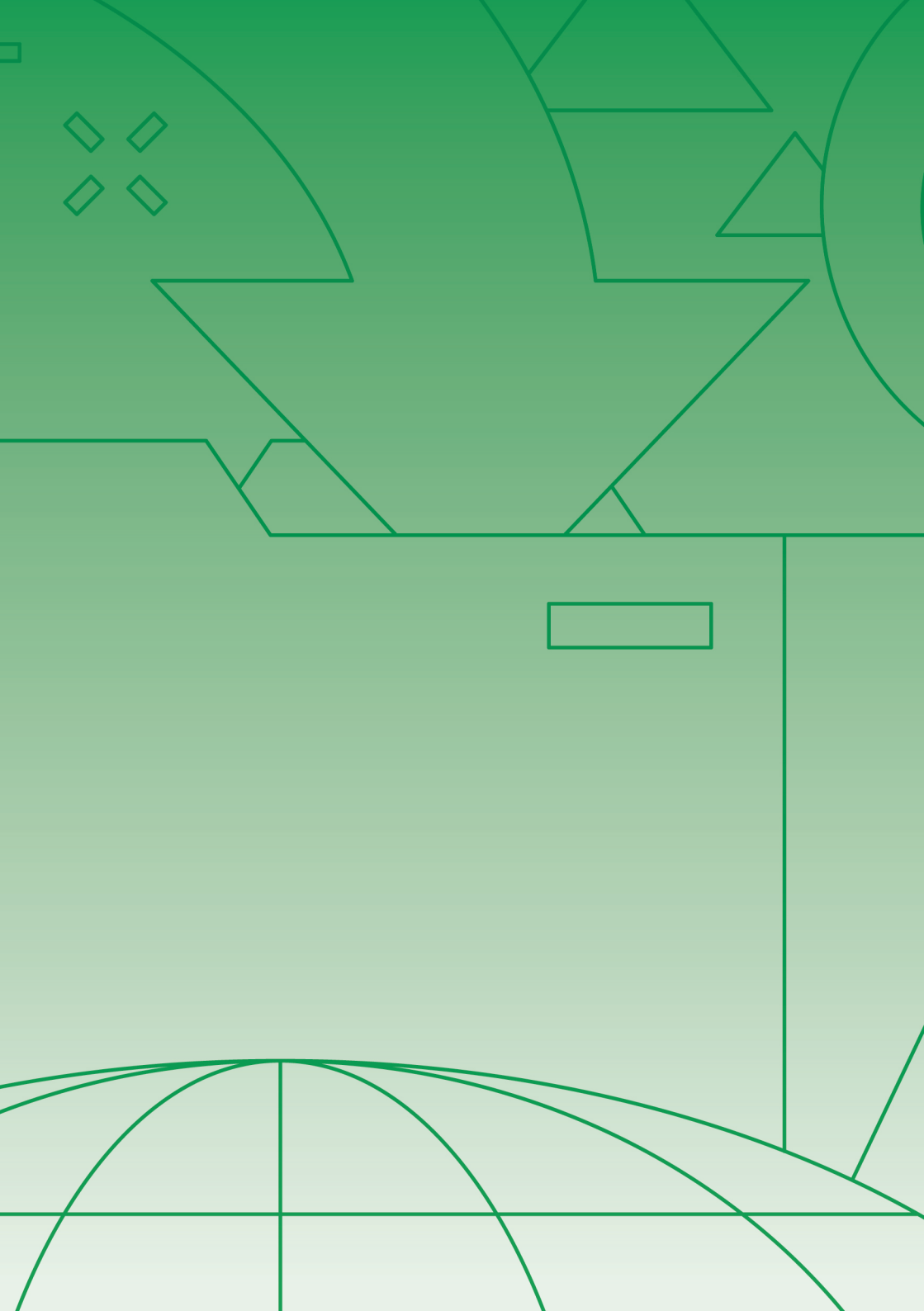
```
source ~/env/bin/activate
```

You should then see a prompt like the following:

```
(env) username@hostname:~ $
```

To leave the virtual environment, run the `deactivate` command. Once you have your virtual environment set up, you can install a Python package with a command like `pip install numpy`. If you need a different set of libraries for a different project, you can create another environment with a different name and activate it when needed.

Happy installing, and above all, don't worry — it's quite hard to mess up the Raspberry Pi OS installation, but if you do, at least those backups you made earlier will get tested out.



Chapter 14

Commanding the internet

Yes, it is possible to get online from the command line. Here's how...

Now that you're getting comfortable at the command line, and perhaps finding it faster for some tasks, you may want to increase the amount of time you spend there, trying to improve your productivity on everyday tasks. Here we survey internet software — a diverse field, covering clients for protocols, specific services (YouTube), and tasks (search), as well as general web browsers, mail clients, and even surprising uses of the venerable Telnet client.

Don't get carried away: command-line alternatives vary from essentials you'll see often used in scripts and Raspberry Pi projects, to less satisfactory alternatives you'd only use when there is no GUI available — so don't plan to switch all of your Facebook and Instagram use to the terminal. We concentrate here on internet interactions which will be useful for your Raspberry Pi project, but sometimes you may be using the command line on Raspberry Pi and just need to look something up on the web, so welcome back to the Internet of Text.

These days, Raspberry Pi is a very capable desktop machine. However, some websites are so dependent upon JavaScript that for all the memory that they hog on Raspberry Pi, you can only access them via Chrome (or Firefox). For others, try the retro world of the command-line web browser.

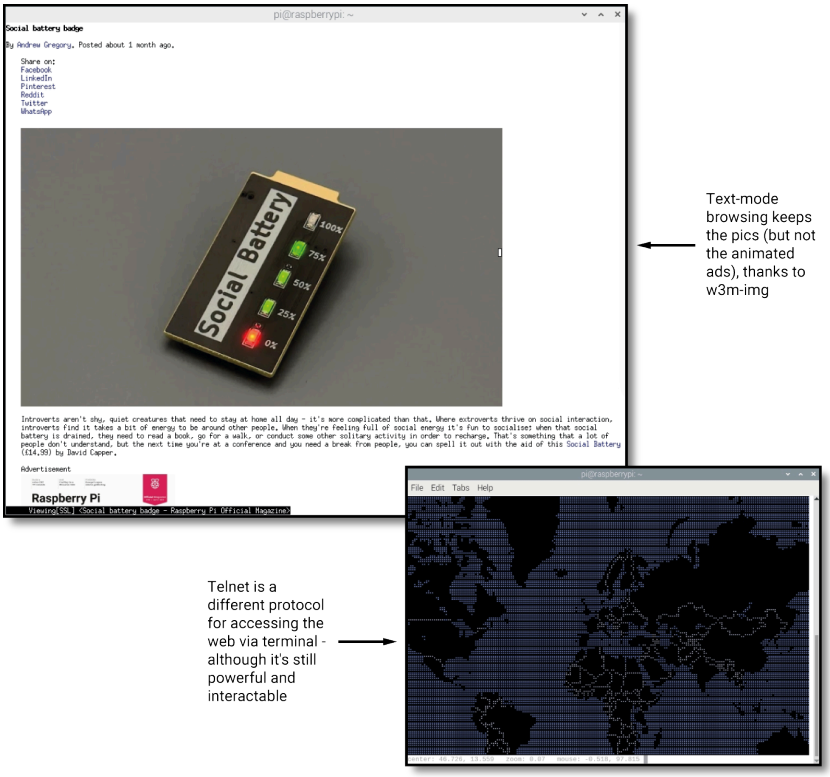


Figure 14-1 Online from the command line

In the early 1990s, if you needed to browse the web from a command-line environment, you probably used a text-mode browser called Lynx. More than a quarter of a century later, it's still in use! If not being able to see any pics makes the web pointless for you, consider the advantages: minimal tracking, no distractions from reading the text, and quicker page loads. Screen reader users and people with mobility impairments can benefit from the simpler keyboard navigation, and as pages can be dumped to stdout, you can pipe it through other command-line applications:

```
lynx -dump https://duckduckgo.com/?q=gpio+bash | tail -n 30
```

Lynx isn't the only choice in command-line browsing — you can install the links2 and w3m packages with `apt install`; the latter is even usable from within the Emacs text editor. Install xterm (an alternative to the default Terminal application in Raspberry Pi OS) and w3m-img, and you can even view images when browsing from the terminal.

Commanding the web

Telnet — short for teletype network — is the protocol developed at the end of the 1960s to provide two-way text communication between computers (install it with `apt install`). It dates from an era without security concerns, and was long ago replaced by SSH as a means of connecting directly to other machines over open networks, but the Telnet client remains useful for interactively querying services such as mail or web for testing, by connecting to the appropriate port on the server, then issuing commands in the service's protocol:

```
telnet example.com 80
GET / HTTP/1.1
HOST: example.com
```

...which (after pressing **ENTER** again) should dump a webpage onto the terminal. It's a basic but useful diagnostic tool.

Surfraw

As we have seen earlier, you don't need a web browser to download files if you use `wget` and `curl`. Another case where specialist commands seem most appropriate is searching. Looking things up from the command line can be done directly with a nifty little program called `surfraw`, which uses helpers called `elvi` to tackle different search tasks — see them all with:

```
surfraw -elvi
```

As a good citizen of the command-line world, surfraw pays due regard to your time and your tendons, and can be called simply with `sr`.

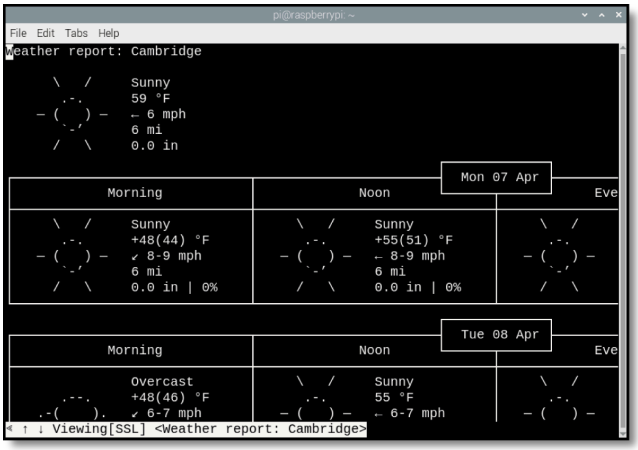


Figure 14-2 Command-line browsing is all about the information — mostly text, but not entirely

```
sr google -results=2 raspberry pi clojure gpio
sr translate peloton
sr gutenberg dickens
sr wikipedia permaculture
sr bbcnews tim peake
sr rhyme orange
```

The results are displayed in your browser, which you can set in **/etc/xdg/surfraw/conf**:

```
def SURFRAW_text_browser /usr/bin/lynx
defyn SURFRAW_graphical no
```

Play button

YouTube videos also won't work from a text mode browser, but it's easy enough to grab them with **yt-dlp**, a fork of **youtube-dl**. You can grab the code from **rpimag.co/ytdlp** and compile it using the information from Chapter 13, *Easy compilation*.

Once you have, you can try downloading a video with:

```
yt-dlp https://www.youtube.com/watch?v=ouhXh8thnQM
```

Each downloaded video is saved as an webm file which, thanks to the built-in video player's (VLC) ability to output video through a choice of libraries, can even be rendered in ASCII through the terminal, using `DISPLAY= cvlc -V aa video.webm`. The `DISPLAY=` before the command makes VLC unable to use the GUI, forcing it into text-mode output (you don't need it if you're connecting over SSH or from the text console).

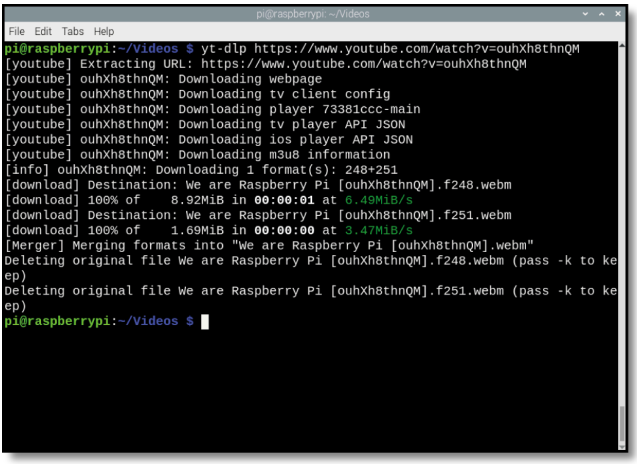


Figure 14-3 No spare memory to open a YouTube link? Download the video to watch later

Communication

Command-line apps are available for other means of internet communication, from IRC to Jabber, and Mutt is a powerful enough email client to keep people (who otherwise don't spend much time in the terminal) using it. Serious work gets done in a shell, but coders keep churning out command-line apps that are just for fun, too.



MAKE EMAIL MORE MANAGEABLE

If you struggle to keep up with email using a webmail interface, challenge yourself to use Mutt for two weeks: the command line may let you finally tame your inbox.

MapSCII is a Braille and ASCII map renderer for your console, using OpenStreetMap data, written by Michael Straßburger (**rpimag.co/mapscii**). To connect from a remote computer:

```
telnet mapscii.me
```

ASCII Star Wars appeared at the end of the last century, and is still available at **asciimation.co.nz**. Ironically, you'll need a modern graphical browser to view it.

Some sites are designed to look good in text-mode browsers. A favourite is **wttr.in**, the weather website. Enter your location, and open in any of the text-mode browsers that we've mentioned; for example:

```
w3m wttr.in/Liverpool
```

Sometimes only words will do. Graphical user interfaces (GUIs) were a great advance, creating an easy route into computers for non-technical users. But for complex tasks, GUIs can block off choices and require a circuitous route even for simple tasks.

Enter the command line: the blinking cursor many thought had faded away in the 1990s. For getting instructions from user to computer — in a clear, quick, and unambiguous form — nothing beats the command line. It never disappeared on UNIX and Linux systems, and thanks to Raspberry Pi OS, a new generation is discovering how it simplifies complex tasks and lets you carry out simple ones instantly.

Build essential skills, including:

- Read and write text files
- Find and install software
- Manage removable storage
- Use Secure Shell (SSH) for remote access
- Write disk images to SD cards
- Browse the web from the command line

If you're not comfortable when faced with the $ prompt, don't panic! In this fully updated book, we'll quickly make you feel at home and get you familiar with the terminal on Raspberry Pi (or any Mac and Linux computer). Updated for the latest Raspberry Pi software, this book has everything you need to get started.

Raspberry Pi PRESS

raspberrypi.com

ISBN 978-1-916868-38-0

UK £10.99
US $13.99