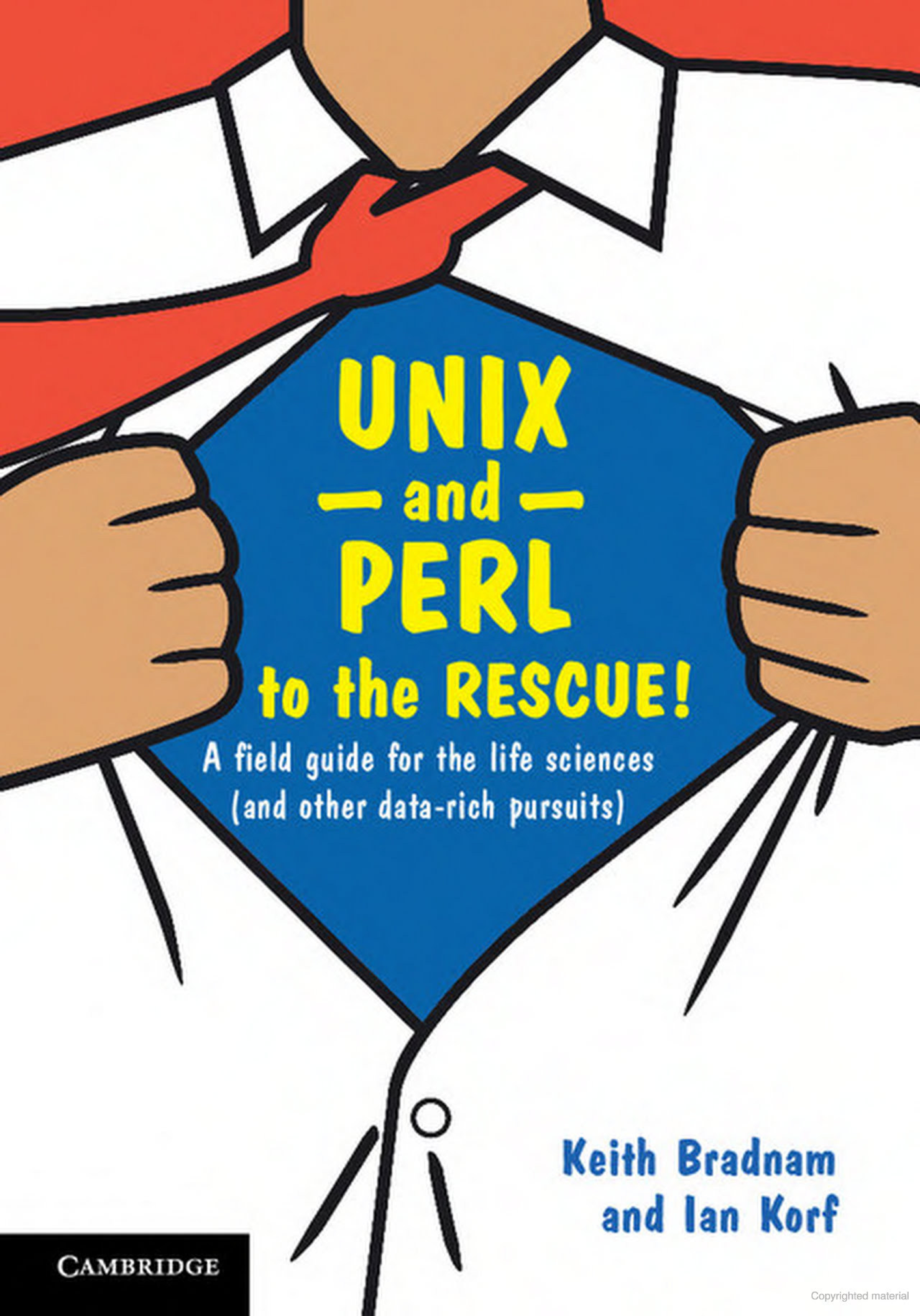


UNIX — and — PERL to the RESCUE!

A field guide for the life sciences (and other data-rich pursuits)

Keith Bradnam and Ian Korf

CAMBRIDGE

Unix and Perl to the Rescue!

A field guide for the life sciences (and other data-rich pursuits)

KEITH BRADNAM
University of California, Davis

IAN KORF
University of California, Davis

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town,
Singapore, São Paulo, Delhi, Mexico City

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9781107000681

First published 2012

Printed in the United Kingdom by the MPG Books Group

A catalogue record for this publication is available from the British Library

Library of Congress Cataloguing in Publication data
Bradnam, Keith.
UNIX and Perl to the rescue! : a field guide for the life sciences (and other data-rich pursuits) / Keith Bradnam, Ian Korf.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-107-00068-1 (hardback) – ISBN 978-0-521-16982-0 (paperback)
1. Science – Data processing. 2. UNIX (Computer file) 3. Perl (Computer program language)
I. Korf, Ian. II. Title.
Q183.9.B73 2012
005.4′32–dc23 2011047927

ISBN 978-1-107-00068-1 Hardback
ISBN 978-0-521-16982-0 Paperback

Contents

Introduction and background 1

and teach the occasional class or two. However, he would give it all up for the chance to be able to consistently beat Ian at foosball, but that seems unlikely to happen anytime soon. Keith still likes Macs and neatly written code, but now has a much harder job finding English puddings.

As a youth, Ian Korf's favorite classes were sciences and his favorite pastime was computer gaming. At the time, you wouldn't have thought that hacking and writing computer games would be very useful skills for a budding molecular biologist. Certainly nobody ever counseled Ian to do so, especially when he was doing it at 2 a.m.! But apparently the misspent hours of youth can sometimes turn out to be worthwhile investments. Ian's first experience with bioinformatics came as a post-doc at Washington University (St. Louis), where he was a member of the Human Genome Project. He then went 'across the pond' to the Sanger Institute for another post-doc. There he met Keith Bradnam, and found someone who truly understood the role of communication and presentation in science. Ian was somehow able to persuade Keith to join his new lab in Davis, California. This book is but one of their hopefully useful contributions.

Acknowledgments

This book evolved from a course that we both teach to graduate and undergraduate students at UC Davis. We are grateful to the students for their patience with us, as this course has evolved quite a bit since we started teaching it. Their feedback, and their enthusiasm for learning Unix and Perl, have made this book what it is. We also would like to thank Nancy Parmalee for helpful suggestions about the index.

Keith would also like to thank his wife Mel for her tireless support and understanding throughout the long book-writing process. He would also like to express profound gratitude to the wonders of caffeine, the relaxation afforded by his iTunes music library, and to the entire nation of Belgium.

Ian would like to thank all his students, past, present, and future. May your adventures take you to lands unimagined, and your skills see you safely home.

This book was written using Apple's excellent 'Pages' word-processing software, with extensive use of Dropbox software by Dropbox Inc. to make the process of collaborative writing a joy. Code examples were written using TextMate by MacroMates Ltd and TextWrangler by Bare Bones Software.

How to use this book 1.2

Or rather, how *not* to use this book

Organization

This book is divided into seven parts (you are currently reading Part 1). You may be impatient to start programming with Perl, but if you don't know any Unix we suggest that you start with Part 3, which will teach you the basics of Unix. When you finish that, you can optionally jump ahead to Part 5, which covers some advanced Unix topics. Or you might just want to proceed to Part 4, which covers all of the fundamentals of Perl. The choice is yours. Of course, if you don't yet have Unix and Perl installed on your computer, then you might want to start with Part 2, which covers how you can get Unix and Perl for your PC.

If you've never programmed, we hope that after learning the 'essential' Perl of Part 4, you will be able to write many fantastic and powerful scripts. More importantly, we hope that you will be able to write scripts that are actually *useful*. For this part of the book, we've tried, where possible, to only ever introduce one new concept at a time. Hopefully this will prevent you from being overloaded with too many new concepts at once. This also keeps chapters short and, mostly, self-contained. For a few topics that have increased complexity, we use two or more chapters to cover all aspects of that topic.

We have strived to make sure there are lots of examples. These are all scripts that we encourage you to copy and try yourself. However, you may still gain much understanding just from reading them. In addition to the examples, Part 4 of this book also features a number of problems at the end of most chapters.[4] You are strongly encouraged to tackle the problems. Ultimately, this it the best way to learn Perl (or any programming language). For each problem we provide a solution,[5] but be aware that one of the famous mottos associated with Perl is:

TMTOWTDI[6] – There's more than one way to do it

We have hopefully provided solutions that are easily understandable, but if you want to solve each problem in a different way then that is great.

The topics covered in Part 4 might be all you ever need to know in order to solve many different problems. However, we go further into the more advanced aspects of Perl in Part 6. The distinction between 'essential' and 'advanced' is somewhat arbitrary. If you finish Part 4 then you should at least have a look at Part 6.

Part 7 covers many different subjects that are not unique to Perl. In general, this is the section that focuses on 'good programming practices.' Most subjects in this part are relevant to many programming languages, though we also include two sections on how to fix broken Perl scripts.

Finally, we should note that we do not cover every aspect of Unix and Perl. The world of Unix is especially vast, and several books would be needed in order to cover

[4] We include some problems in the Unix section too, but not as many.
[5] Included in an appendix.
[6] Some people pronounce this 'Tim Toady.'

the myriad number of Unix commands you could learn about. Likewise, we do not cover every feature or function available in Perl. However, we strongly feel that this book covers all of the basics (and much more besides). Readers are therefore encouraged to use this book as a launch pad for a journey into a much wider world of programming. If you develop a hunger for learning about new Unix commands, Perl functions, and even new programming languages, then dare to venture beyond the confines of this book. You will be rewarded!

Style conventions

Each chapter has a main heading and a subheading. The subheadings are one area where we have tried to inflict our pun-tastic sense of humor on you.[7] We will often include both Unix and Perl examples, which you should attempt to follow. The Unix examples will include simple instructions of Unix commands that you should type, whereas the Perl examples will contain complete scripts, accompanied by line numbering. E.g.:

Example 1.2.1

```
1.   #!/usr/bin/perl
2.   print "The shortest script in the world?\n";
```

The line numbers are just there so we can refer to them in the text. You are not meant to type the line numbers! Following just about every example will be a section that tries to explain what the point of the example was. For the Unix examples, this will be a section titled *Explanation*, but for the Perl scripts it will be a line-by-line breakdown of how the script is working. E.g.:

Understanding the script

Line 2 contains a simple `print` statement.

In addition to having worked-through examples, we will also set problems that you should try to solve. Where appropriate, answers will be provided, but we encourage you to try solving the problems without looking at the solution.

Hopefully you will have noticed that we use a fixed-width font for writing any Perl or Unix code. This will be done for complete scripts and even when we mention a single Unix command or Perl function within a sentence. E.g., we might mention that the Unix command `sed` shares similarities with Perl's substitution operator (`s//`).

Sometimes we will show fragments of Perl scripts, just to illustrate a point or to demonstrate the syntax of a command. We do not include line numbers for these examples, and they are not intended to be run as complete scripts. E.g.:

```
my @array_A = @array_B; # copying an array
```

[7] Footnotes, like this one, are another place where you might find occasional diversionary comments on matters which may not be entirely related to Unix and Perl. E.g., did you know that there are no words in the English language that rhyme with the word 'orange'?

Occasionally, we will want to shout something at you because it is *so* important, and the world will cease to exist if you fail to understand the critical point we are making. E.g.:

The world will cease to exist if you fail to understand the critical point we are making!

Any time you see something written in this style, you should probably re-read it several times and remember that we will be not-inconsiderably displeased if you fail to remember our advice!

Installing Unix and Perl 2

run Linux. Because the underlying code used by all Linux systems is freely available, many companies have packaged together slightly different versions of the OS[3] that you can download for free. This very attractive price point, coupled with the fact that it is possible to run Linux without having to install anything on your computer's hard drive (see the next chapter) means that Linux is a great solution for PC owners wanting to work through this book.

Unix vs. Linux: part 1

When we say that we want to teach you Unix, we are not talking about learning the entire OS.[4] Instead, we want to teach you about a few of the Unix commands that you will have to type into a program known as a *terminal* (which we will explore in Chapter 3.2). There are probably a few thousand Unix commands, but we are only aiming to teach you 20 or so of the most essential ones. All of the Unix commands that we will teach you are also available on Linux. This means that all we need from a Unix or Linux OS is the ability to open a terminal program and run some Unix commands. All other differences between Unix and Linux are not important for this book.

Unix vs. Linux: part 2

You do know that there is an exception to every rule, right? While we are confident in saying that there are no important differences between Unix and Linux, you should be aware that there are still differences, both between Unix and Linux and between different versions of Unix/Linux. These differences can mean that some of the commands that you type *may* produce slightly different output to what we show in this book. Sometimes this is because one OS might have a newer version of a Unix command when compared to another OS.

This means it would be impossible to write this book in a way that makes it fully compatible with *all* possible Unix and Linux variants, so we have written this book using Apple's version of Unix.[5] We are 99.9% sure that every Unix command we mention will be available on whatever form of Unix or Linux you use, but bear in mind that the output of our Unix examples might sometimes look different to yours.

Learning Perl without learning Unix?

If you are using a Windows PC, then you might not want to install Linux and you might just want to learn Perl. That's fine, but bear in mind that our Perl examples are written from a Unix point-of-view, so we will show examples of how to run Perl scripts from the perspective of a Unix system.

[3] Such packages are known as *distributions* and usually contain a slightly different mix of software tools and sometimes a different GUI.

[4] Otherwise you could argue that learning to use a Mac is a way of learning how to use Unix.

[5] We have chosen Apple because we both use Apple computers for our work (including running Unix commands and writing Perl scripts).

Installing Linux on a PC 2.2

There are virtually a dozen solutions

There are four main ways you can install a Linux OS on your Windows PC:

(1) Install Cygwin. This provides a Linux-like environment on your PC; it is free to download.
(2) Run Linux from a CD-ROM or install on a bootable flash drive. This is a good solution for people who don't want (or do not have permission) to modify the contents of their hard drive.
(3) Install a full Linux distribution on your computer, either as a replacement for Windows or as a dual-boot option.
(4) Run Linux by using *virtualization* software. There are many software packages that will allow you to effectively install an OS *inside* another one.

We will discuss each of these options in a little more detail in this chapter, though it is beyond the scope of this book to provide detailed installation instructions, not least because any instructions we could provide would quickly become out of date (the world of Linux moves very quickly). Bear in mind that the internet contains a lot of information about installing Linux on PCs.

Linux distributions

If you didn't already know, you should be aware that there are many different versions of Linux in existence. They differ in many respects, but the core functionality is very similar no matter which one you choose. The web site www.livecdlist.com lists most of the popular variants that are out there and provides a good starting point for choosing a distribution. Fashions come and go in computing, and it is likely that this list will look very different in a few years' time. One of the most popular full-featured Linux OSs out there at the time of writing is Ubuntu (available at ubuntu.com). For the purposes of working through this book, any of the popular distributions will be fine, but if you want to install Linux on a CD-ROM or flash drive you might want to choose a distribution that requires less space (see the relevant section below).

Installing Cygwin

It is important to note that Cygwin isn't a true form of Unix or Linux. It is software that will result in you having a terminal window through which you can use many Unix programs (including Perl). There are some differences between Cygwin and other types of Unix, which may mean that not every Unix example in this book will work exactly as described, but overall it should be sufficient for you to learn the basics of Unix. At the time of writing, Cygwin is free and is under active development. You can download it from www.cygwin.com.

Running from a CD-ROM or flash drive

Storage capacities of USB flash drives continue to increase (and prices decrease), which means it is possible to store entire OSs on them. It is also possible to boot your computer from a USB drive and run the OS that you have installed. Because not everyone has a large-capacity flash drive, and because some people would like to run Linux from a CD-ROM, there has been a demand for lightweight Linux distributions which omit some of the less essential parts in order to fit on a flash drive or CD-ROM. Linux distributions such as Damn Small Linux (www.damnsmalllinux.org) go so far as to fit an entire OS into less than 50 MB of space. Other popular versions of Linux which are also compact are Slax (available at www.slax.org) and Puppy Linux (www.puppylinux.com).

The obvious advantages to these methods are that you don't need to add anything to your computer and you can easily take your Linux OS anywhere you go. Some of these solutions will still require you to boot your computer from the flash drive or CD-ROM (meaning that you can't use Windows until you reboot). However, some of these solutions can also be run without restarting your computer, meaning that you can access Linux within a single window.

Install Linux

If you try out the method above and discover that you like Linux, you may want to make it your main OS, or at least have it available to run alongside your Windows OS. It is very common to find 'dual-boot' machines, which means you can choose which OS you want to run from a menu after turning the machine on. Setting up Linux in this manner will require sufficient free space on your hard drive and you will also need administrator privileges on your computer. Follow the instructions provided by your Linux distribution. Alternatively, rather than have a dual-boot system, you can make Linux your main OS and run Windows using virtualization software.

Virtualization software

By installing suitable software, virtualization effectively allows you to run one (or more) different OSs within your main OS. It is very commonly used to run Linux as a *guest* OS within a Windows OS, and vice versa.[6] Most modern computers will have sufficient hardware to do this. Some virtualization software is free to download. This is another fast-moving field within the software industry and it is therefore hard to make specific suggestions as to which software to use. Some popular virtualization solutions that are currently available include Microsoft Virtual PC, VirtualBox (free from Oracle), VMware Player, and Parallels Workstation. However, please note that there are also many other products available and we do not endorse any one of these products.

[6] It's also very common to see virtualization software used in such a way that one computer runs older versions of its own OS (e.g., running Windows XP from within Windows 7).

Installing a code editor 2.3

So I installed Perl and Linux, can I start writing code yet?

The majority of this book will teach you how to write Perl scripts, and you will therefore need something to write them with. Like other scripting languages, Perl scripts are just text files and can therefore be written with any software capable of producing a plain-text-format file. Note that *plain text* specifically means text that is devoid of formatting. All OSs come with basic text editors that are capable of producing plain-text files.[7] You should *not* use these editors. Nor should you use a fully fledged word-processor program such as Microsoft Word. This point bears repeating:

Do not use a word processor to write code, it will cause stress and grief!

You should instead use a program that is specifically designed to write code. Such programs are known as *text editors* or *source code editors*, and include a number of features that will greatly help you as you learn to program in Perl. The most important reason for using a code editor is that they already know about the syntax of Perl (as well as many other programming languages) and will change the color of what you write in a process called *syntax highlighting*. A simple analogy would be to imagine that you could write a sentence in English and have all the verbs and nouns automatically change to red or blue. If you mistyped the name of a verb, then it wouldn't turn red, and this would give you instant visual feedback that there was a problem. As you will quickly learn, fixing bugs in code can be a troublesome task, so anything that helps you find bugs as you write them is to be welcomed.

Apart from syntax highlighting, code editors have many other useful features and it is essential to use them when writing any code. There are many free ones available and you should ideally try out several to find one that works for you. They all have slightly different combinations of features and some are cross-platform, whereas others will only be available for Macs, PCs, or Linux OSs. As a starting point, we would suggest Notepad++ for Windows, TextWrangler for Macs, and Gedit or jEdit for Linux. Once again, we are not trying to endorse any particular piece of software. As you start to type a lot of code, the relationship between you and your code editor becomes very important, and it is highly recommended that you 'test drive' other editors.[8]

[7] Typically, this would be TextEdit on a Mac and Notepad on a Windows PC.

[8] Wikipedia has a very detailed page that compares the features of various text editors: http://en.wikipedia.org/wiki/Comparison_of_text_editors.

Introduction to Unix 3.1

No mouse required!

By this point you should have a computer that runs a version of Unix or Linux. Everything we do in this part of the book will involve *typing* commands using a program known as the *terminal* (more on that in the next section). Unix contains many hundreds of commands, but we only need to learn a small number in order to achieve most of what we want to accomplish.

You are probably used to working with programs like the Apple Finder or the Windows File Explorer to navigate around the hard drive of your computer. Some people are habituated to using the mouse to move files, drag files to the trash, etc., and it can seem strange switching from this to typing commands instead. Be patient, and try – as much as possible – to stay within the world of the Unix terminal. We will teach you many basics of Unix, such as: renaming files, moving files, creating text files, etc. and you may sometimes be tempted to resort to doing this without using Unix. Initially it will feel wrong to do something as simple as moving a file from one folder to another by typing a command. Stick with it and it will start to become second nature. Learning to do things by typing commands also gives you a back-up plan if your mouse breaks!

Throughout this part of the book we will provide lots of Unix examples that you should also type yourself. Please make sure you complete and understand each task before moving on to the next one. We will sometimes show the output of running various commands. In some cases your output will look different to our output because it is very unlikely that any two filesystems will be identical (even on computers with the same OS). Hopefully, though, you will be able to follow all of these examples without getting too lost.

One final note: Throughout the remainder of this part of the book we will refer to Unix over and over again. Every time we mention the U-word, you can equally think of the L-word (Linux). From the perspective of what we are trying to teach you, the two are synonymous.[1]

[1] If this bothers you, then feel free to buy our special 'Linux-edition' of this book. It is identical, except we change all mentions of Unix to Linux.

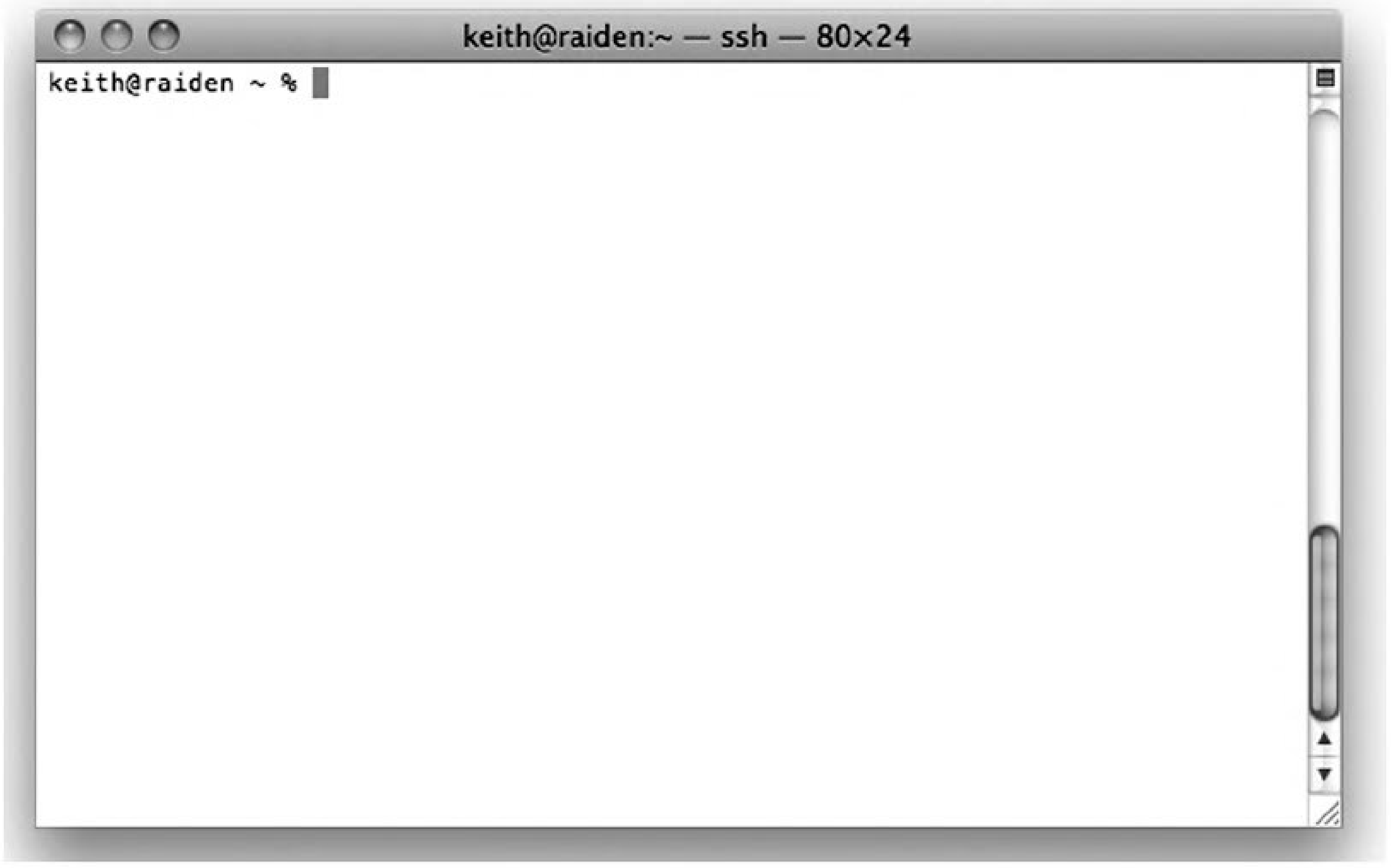
The Unix terminal 3.2

A window into a wider world

A 'terminal' is the common name[2] for a type of program that does two main things. It allows you to send typed instructions *to* the computer (i.e., run programs, move/view files, etc.) and it allows you to see the output that results *from* those instructions. Historically, computers did not have any form of GUI, so the only way of interacting with them was by typing commands to do everything.[3] The keyboard was the only form of input and a single monitor screen was the only form of output. In modern-day OSs, the terminal will be run as just one of many different applications; some people refer to terminal applications as terminal *emulators*.

You can count on all Unix/Linux OSs to have a terminal program, and it is common for any such program to include 'term,' 'terminal,' or 'tty' as part of its name – e.g., on Apple computers the default terminal application is simply named Terminal. Bear in mind that all OSs will also allow you to download alternative terminal programs. These will offer you different degrees of customization as well as slightly different features. However, for the purposes of this book, the differences between any two terminal applications are trivial.

After launching the terminal program, you should see something that looks a bit like this:



```
keith@raiden ~ %
```

This is the standard Apple terminal program. Yours might look very different,[4] but there should at least be some text inside the terminal window, and perhaps a blinking

[2] Also known as a 'term' or a 'tty.'

[3] Of course, there is the whole 'pre-keyboard' era of punch cards and punch tape as forms of data input. But enough with the history lesson.

[4] It is fairly common to see terminal programs use a two-color scheme of either: black on white, white on black, or the *Matrix*-esque green on black.

cursor. The text might just be a single character such as a $, %, or # symbol, or it might include other information such as the name of your computer or your login name.

Customizing your terminal

Before we go any further, you should note that your terminal program will very likely let you alter the appearance of the terminal window. If you explore its options/settings/preferences menu you will probably be able to do things like change the default colors, style of font, and size of font. Initially it might be better to stick with the default settings until you are comfortable using the program, but at some point you should set up your terminal so it is to your liking.[5]

Note that you can resize terminal windows, or have multiple windows open side by side. Some terminal applications will also let you open multiple tabs within a single window. There will be many situations where it will be useful to have multiple terminals open and it will be a matter of preference as to whether you want to have multiple windows, or one window with multiple tabs (there will usually be keyboard shortcuts for switching between windows or tabs).

Before we go any further you might also want to check what keyboard commands are used to close windows or tabs, just so you don't *accidentally* do that.[6] For much of this part of the book you will *only* need to use the terminal, so feel free to resize it to its maximum window size. This might also help you avoid the temptation to start moving/renaming files by using your OS's file browser. Before we proceed with learning our very first Unix command, let's reiterate that last point.

Do not use your mouse. In the land of the terminal, the keyboard is king!

[5] Terminal applications will use a *fixed-width* font for the default font. There are good reasons for this and you should probably *not* change it to any non-fixed-width font (e.g., Times, Arial, etc.).

[6] Closing a terminal window will often, but not always, stop the program or command that you were running. If your program had been running for a day and was just about to give you the answer to life, the universe, and everything else – then you will have to wait another day for the answer.

The Unix command prompt 3.3

We command you to read this section

Hopefully your terminal window already contains some text. As mentioned in the last chapter, this text might include your login name[7] or the name of the machine you are using. The text traditionally ends with a punctuation character of some kind (most commonly a $ or % sign). Collectively, this text is known as the *command prompt*.

Example 3.3.1 It's time for your first interaction with the world of Unix. After you make sure that your current terminal window is selected, take a deep breath and press the enter key on your keyboard.

Explanation

Congratulations! You have just interacted with your Unix terminal. Hopefully the world didn't end and your computer is still intact. Most importantly, you should have noticed that the text that was on the screen *before* you pressed enter has now been duplicated on a new line. Every time you type any Unix command and press enter, the computer will attempt to follow your instructions and then, when finished, return you to the command prompt. Sometimes you might have to wait a while for a program to finish before the command prompt returns, but once you see the prompt, then you know you are free to type your next command. Some forms of Unix provide a blinking cursor, which makes it a bit easier to focus your eye on where you can type. Usually, a new terminal window will have the current command prompt at the *top* of the window. But if you try pressing enter 20–30 times, the current command prompt will get moved to the bottom of the window.

Examples of command prompts

Depending on what version of Unix you are using, your command prompt might also be set up to include the name of the current directory.[8] This might change as you navigate to different directories on the computer (which we will be doing in a few chapters' time). Therefore, don't be surprised if the text that makes up the command prompt changes from time to time. Command prompts can also be customized to include a lot of other information. Here are a few examples of what some command prompts can look like:

[7] Also called the 'user name.' It's entirely possible to have one account name that you use to login to your computer (e.g., 'keith') and then have a different Unix login name (e.g., 'themaster').

[8] Directories are the same thing as what you might think of as 'folders' when using a graphical file manager.

Prompt	Description
`$`	A single-character prompt
`%`	Another common single-character prompt
`bash-3.2$`	The default prompt on Mac OS X. This prompt includes the name and version number of something called the *shell*. More of that in a later chapter.
`nigel@stonehenge$`	A prompt that includes both details of the user name (nigel) and the computer name (stonehenge). It's very common to see this type of information included in the prompt.
`nigel:/home %`	A prompt that contains the user name as well as the name of the current directory (/home). We'll explore the syntax of directory names later.

Because of this huge diversity in command prompts, we will stick with using a single dollar sign for all examples which include the command prompt. If we show you the following:

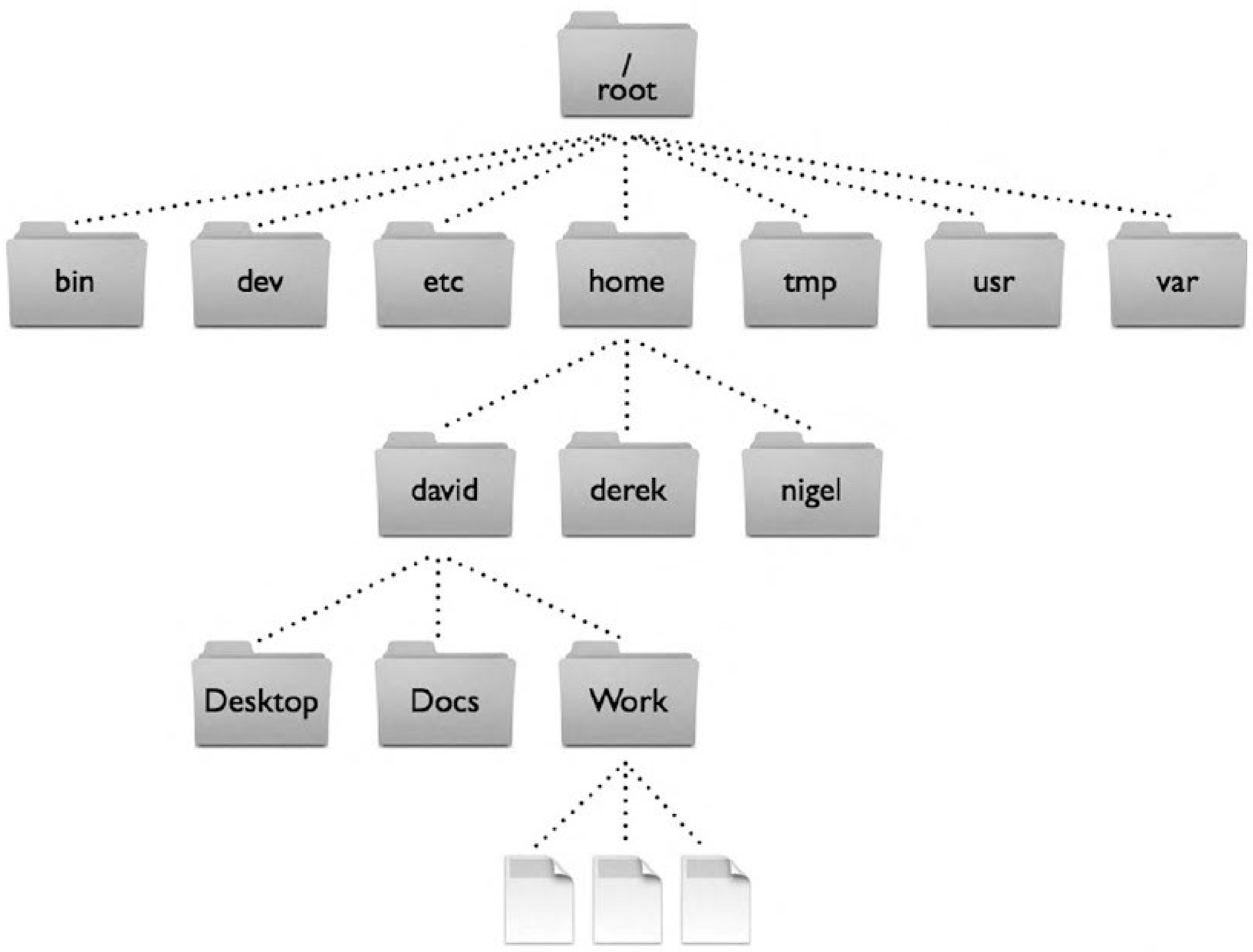
```
$ ls
```

This should be interpreted as *type the Unix command '`ls`' at the command prompt.* And if you're wondering what the ls command does, then you only need to look at the next chapter!

The hierarchy of a Unix filesystem 3.5

This will be a root-and-branch review

Looking at a list of directories from within a Unix terminal can often seem confusing. But bear in mind that these directories are exactly the same type of folders that you can see if you use your computer's graphical file management program. A tree analogy is often used when describing computer filesystems because of the branching nature of the directory structure. Like a tree, a Unix filesystem has roots, or more specifically, it has *a root*, which is represented by a single forward-slash character (/).[14] The analogy with the tree starts to break down a little as it is common to refer to the root level as the 'top' of the directory structure – think of it as an inverted tree if it helps. From the root level (/) there are usually many (10–20) top-level directories. A small number of these directories will be present on all Unix systems, but there will also be many directories that are specific to different types of Unix. Here is a fictional example of what *part* of a Unix tree might look like:



In this example we show seven top-level directories below the root level.[15] These directories may seem to have strange names, but you don't really need to know what they are for. The one thing to note from this schematic is that there is a 'home' directory which in turn contains the home directories of three users ('david,' 'derek,' and 'nigel'). We then show that David's home directory contains more directories, one of which ('Work') contains some files. This schematic is highly simplified – a full Unix

[14] On a standard Windows OS the equivalent to the root level would be C:\.

[15] These are the directories that occur on nearly all Unix systems.

filesystem may contain several hundred directories, and many thousands of files. Note that we shall return to this fictional filesystem in the next few chapters.

Directories that exist inside other directories, are often referred to as subdirectories.

Example 3.5.1 If you want to see what your own root level looks like you can simply run the `ls` command and tell it that you want to list the contents of the root directory (represented by a single forward-slash character):

```
$ ls /
Applications      System      cores      mach_kernel      tmp
Developer         Users       dev        net              usr
Library           Volumes     etc        private          var
Network           bin         home       sbin
```

Explanation

This is the listing of the root directory on a computer running Mac OS X. Note that it has a directory called 'home' (like many Unix systems) but the actual home directories of users are stored as subdirectories within the 'Users' directory. There is often a lot of variation as to where different Unix systems keep home directories,[16] and in the next chapter we will learn how to find out where your own home directory is located.

Navigating the filesystem tree

If we briefly return to our fictional Unix filesystem, we should note that the directory called 'david' contains three subdirectories and is itself a subdirectory of the 'home' directory. If we wanted to copy or move some files that are in David's 'Work' directory to Nigel's home directory, we can trace a path that would navigate *up* three levels in the directory tree (→ Work → david → home), and then navigate down one level (→ nigel). If this concept of going up and down various branches seems intuitive to you, then great! If it doesn't, then it might help you to start thinking about your filesystem in this way. This means that if you wanted to copy a file from David's 'Work' directory to his 'Docs' directory, you actually have to go up one level and then down one level (rather than go directly across).[17]

[16] Though it is very common to name the actual home directory after your login/user name.

[17] Of course the concepts of 'up,' 'down,' or 'across' are somewhat misleading. All directories, programs, and other files exist as series of binary 1s and 0s on a magnetic disk or other storage medium. However, these concepts can sometimes make it a lot easier to understand how to use many different Unix commands.

Finding out where you are in the filesystem 3.6

You should learn to use pwd PDQ

As we have already mentioned, there may be many hundreds of directories on any Unix machine. So how do you know which one you are in? In Unix, the current directory you are in is known as the *working directory*. To find out where you are, you can run the Unix command `pwd`, which will *p*rint the *w*orking *d*irectory, and this is pretty much all this command does. If you have opened a new terminal window you will normally be placed in your home directory:[18]

```
$ pwd
/Users/nigel
```

In this example we are in the home directory of a user called ‘nigel,’ who has his home directory located as a subdirectory of the ‘Users’ directory. Conversely, ‘Users’ is ‘above’ the level of ‘nigel’ and Unix would refer to this directory as the *parent* directory of ‘nigel’.

Slashes separate parts of the directory path

As we have just seen, Unix uses forward-slashes to separate out the various parts of a directory location. In the above example we can see that ‘nigel’ must be a subdirectory of ‘Users’ because the two are separated by a single forward-slash character. Collectively, the set of file and directory names that are combined with forward-slashes is known as a *path*. A single path will always specify some unique location in the filesystem.

If a path *starts* with a forward-slash character, then this is the same thing as the single forward-slash that represents the root level. In the above example, because it starts with a forward-slash, we can infer that ‘Users’ must be a directory one level beneath the root directory. We shall return to the issue of navigating paths in the next chapter.

Remember to use `pwd`!

As you become more familiar with Unix, you will find yourself trying to switch between different directories in the filesystem (which we will learn how to do in the next chapter). The more you move around, particularly as you start using multiple terminal windows, the more likely it is that you will get ‘lost’ in the filesystem. When you start running more complex commands (or Perl scripts) a common reason for them not appearing to work is that you are not in the correct directory. As you start to learn Unix, it is good to get into the habit of frequently running `pwd` to check that you always know where you are.

[18] You can configure Unix to start your terminal sessions somewhere other than your home directory and this is sometimes useful, but 99% of the time a new terminal will place you in your home directory.

How to navigate a Unix filesystem 3.7

It's time for a change

We have previously seen how the `ls` command allows us to 'look' at the contents of any directory in a Unix filesystem. It is entirely possible, and sometimes desirable, to perform actions with files or folders that are in different directories with respect to your current working directory. However, we frequently want to change directories so we are in the same directory as some file or program. We can do this using the `cd` command (*c*hange *d*irectory). Let's return to our fictional Unix filesystem and see how the user Nigel would change directory from his home directory to the temporary directory 'tmp'.[19] Feel free to repeat these steps from your own home directory:

```
$ pwd
/home/nigel
$ cd /tmp
$ pwd
/tmp
```

We start by confirming that we are in Nigel's home directory by running the `pwd` command. We then use the `cd` command to change to /tmp. Note that the `cd` command does not give you any output or feedback after you run it. If we knew what files we were expecting to see in /tmp then we could run the `ls` command to check that this is the intended location. However, it is always possible that two different directories could contain identical contents. That is why we run the `pwd` command again, to confirm that we are really in /tmp.

Changing directories in multiple steps

Let's imagine that we want to change directory to the 'home' directory and then change to David's 'Work' directory. We will omit the `pwd` confirmation step from now on:

```
$ cd /home
$ cd david/Work
```

There are two things to note here. First, notice how that second `cd` command does not start with a forward-slash. You only need to include a forward-slash at the start of a Unix path when you want to start navigating from the root level of the filesystem. Hopefully, you also noticed that performing this in two steps is a little pointless. If you just want to navigate from directory 'A' to directory 'B,' you can always do that in one step:

```
$ cd /home/david/Work
```

[19] The 'tmp' directory is used for storing various temporary output from programs. Be aware that it is automatically emptied at periodic intervals, so you should not use it to store important files.

Can't change directory?

Remember that many Unix systems are case-sensitive. This means that if we had typed 'David' instead of 'david' we might have seen an error message like the following:

```
$ cd /home/David/Work
cd /home/David/Work: No such file or directory
```

We would also see this type of error message if we had misspelled any part of the path, or missed out any of the forward-slashes. If we mistakenly typed something like:

```
$ cd home/nigel
```

instead of:

```
$ cd /home/nigel
```

without the leading forward-slash character, Unix will assume that we want to change directory to a subdirectory of our current directory called 'home'. It is important to be able to appreciate the difference between the two commands shown above. The second command will only ever specify a *single* location on the filesystem, as there can only be one directory called 'home' at the root level of the computer. In contrast, the first command could potentially work in multiple places as there could be another 'home' directory somewhere else in the filesystem.[20]

Changing to the parent directory

One of the most common uses of the `cd` command is to navigate to the directory above the one you're currently in – e.g., you want to change from the 'david' directory to 'home' or from 'home' to '/'. This is straightforward if you know what the directory above you is called. However, you may have forgotten where you are and you may not want to keep on running the `pwd` command to find out. Luckily, you can simply tell the `cd` command to go 'up' one level by using the following syntax:

```
$ cd ..
```

Two dots (without a space) are used by Unix to refer to the parent directory. You can also use this with the `ls` command to list the contents of the parent directory:

```
$ ls ..
```

If you wanted to navigate up *two* levels then you simply need to include a forward-slash between two sets of double dots:

```
$ cd ../..
```

The use of the forward-slash is consistent with what we have already seen about Unix paths, and the forward-slash acts as a delimiter that separates out different levels in the overall directory hierarchy.

[20] This is analogous to the fact that every US city can potentially have an address called '1600 Pennsylvania Avenue,' but within a *single* city it is likely that there is only going to be one address with that name.

layout of your own filesystem. Changing to a new computer can sometimes be a little like moving to a new town. You have to learn where everything is all over again and it takes time to learn the fastest way of getting from 'A' to 'B.'

One final warning for this section. It's entirely possible to do pointless things with the `cd` command, such as:

```
$ pwd
/home/david/Desktop
$ cd ../../../home/david/Docs
```

or similarly:

```
$ pwd
/home/david/Desktop
$ cd /home/david/Docs
```

In these examples we needlessly navigated all the way to the top of the filesystem and then back down again in order to just change directory to a directory that is at the same level as where we started. If you want to do all of that unnecessary typing then just be aware that Unix isn't going to stop you.[22]

[22] By extension, you can even navigate up and down the filesystem and end up right back where you started. We bet you're going to try that right now, aren't you?

Working with your home directory 3.9

(The) home (directory) is where the heart is

What is a home directory?

Of all the directories on a Unix filesystem that you will work with, the home directory is probably the most important. This directory serves the same purpose as the 'My Documents' folder on a Windows computer, and it is where you will store various files that are owned by you.[23] Reflecting its special status, it has a few important properties that set it apart from other directories. It is common for your home directory to be named after your real name or your login name (of course, the two might also be the same). This means that if you have two or more users on your computer who have the same name, they will need to use different names for their home directories.

By default, new terminal windows should always place you inside your home directory. Of course, you can always confirm the location of your home directory (in a newly opened terminal window) with the `pwd` command:

```
$ pwd
/home/nigel
```

Finding your way home

One of the most common acts of 'directory navigation' that you will perform is to return to your home directory (from wherever you were). If you know where your home directory is, you can simply use the `cd` command to go there:

```
$ cd /home/nigel
```

Because you will want to perform this action over and over again and because you might not always remember where your home directory is,[24] Unix provides several useful shortcuts:

```
$ cd ~nigel
```

This command should be read as 'change directory to the home directory of the user called 'nigel.'' The ~ character (known as a tilde) is used by Unix to refer to a home directory. Note that you can use this syntax to navigate to directories beneath the level of your home directory – e.g., the path `~david/Docs` would refer to the 'Docs' subdirectory of David's home directory. Additionally, you can use this syntax to navigate to, or list the contents of, *another* user's home directory:

```
$ cd ~derek
$ cd ~david
$ ls ~derek
$ ls ~david/Work
```

[23] These files will hopefully end up including the many Perl scripts you will be writing!

[24] On large, Unix-based networks, it's entirely possible that your home directory may change location occasionally as it might be moved from one disk to another by your system administrator.

If it is *your* home directory that you want to go to then you can save even more time by omitting the user name altogether:

```
$ cd ~
```

A tilde on its own will always be understood by Unix to refer to your home directory. If we want to save ourselves even more typing, we can take home directory navigation to its extreme:

```
$ cd
```

If you don't provide any other information to the `cd` command, then it will take you to your home directory. This is the format of the command that you will end up using the most. You shouldn't proceed any further without first trying to get lost somewhere in the labyrinth of your own Unix filesystem, before safely returning home with the `cd` command.

Problem 3.9.1 Try to get yourself 'lost' in your filesystem. Change directory to the root level and then start navigating to a different directory at the root level. Check that you can find your way home by simply typing `cd` and then confirming your location with `pwd`.

The Unix shell 3.10

It's time to come out of your shell

What is the shell?

The shell is a command-line interpreter that lets you interact with Unix. You might be thinking that this sounds an awful lot like the terminal, but the two are very distinct. A terminal is like a web browser. There are a lot of web browsers, and they all let you interact with the internet. Similarly, there are a lot of terminal programs, and they all give you a command-line prompt to issue commands and observe the output of those commands.

The shell takes what you type and decides what to do with it. Did you want to run a program? Assign a variable? Autocomplete the name of a file?[25] Pipe the output from one program to another? The shell is actually a scripting language somewhat like Perl. It is not as powerful as Perl, but for some simple tasks a shell script is sometimes more convenient and appropriate. In this book we only touch upon shell scripting, because we prefer to do our programming in a more fully featured language.

Your default shell

Unix is a very flexible OS, and it is therefore not surprising that there is more than one kind of shell.[26] Here is a list of the most common shells:

- *Bourne shell* – commonly known as `sh`. Named after its creator, Stephen Bourne, this shell has remained a popular default shell ever since its original development in 1977.
- *C shell* – known as `csh`. Developed shortly after the Bourne shell, it quickly gave rise to a related shell called the TENEX shell or `tcsh`. The latter shell contains everything in `csh` plus some other useful features such as *command-line completion*. This is something that we will introduce you to in a few chapters.
- *Korn shell* – known as `ksh`. Developed by David Korn in the early 1980s. It includes many features of `csh` and is backwards compatible with `sh`.
- *Bourne-again shell*[27] – known as `bash`. It is widely used and is currently the default shell on computers running Mac OS X. It was developed a decade after `sh`.
- *Z shell* – known as `zsh`. This is the newest of all the shells mentioned so far, and is gaining in popularity. Like most newer shells, it incorporates various elements of all of the other shells that have gone before it, but it also includes new features such as spelling correction (rare among Unix shells).

For the purposes of this book, we only use a small subset of a shell's capabilities, so the differences among shells are minor. For simplicity, use whichever shell is the

[25] We have not discussed auto-completing or tab-completing yet, but it is one of the more useful features of a shell.
[26] Programmers become evangelical about languages, OSs, editors, and to no surprise, shells also.
[27] Unix developers love nothing more than a good pun.

default on your system (we will show you how to find out what your default shell is in the next chapter). There are a few important differences among shells, and we shall cover these differences as and when they arise in the remaining chapters of this part of the book. Bear in mind, however, that it is always possible to change shells (either temporarily or permanently), so you shouldn't feel chained to whatever your default shell is.

Introduction to command-line options 3.12

We command you to check out these options

So far we have only introduced you to a small handful of Unix commands and we have shown you how to run these commands to achieve their default behavior. Usually, the default behavior of a command is all we want, but sometimes we would like to modify the behavior and/or output of the command. For many Unix commands we can produce alternative output by specifying what are known as *command-line options* when we run the command.

Revisiting the `ls` command

Let's assume that our user Nigel has just bought a new computer. The first thing he wants to do is see what the root level of his hard drive looks like.[30] He opens up his terminal application and runs the following command:

```
$ ls /
Applications   System        cores    mach_kernel      tmp
Developer      Users         dev      net              usr
Library        Volumes       etc      private          var
Network        bin           home     sbin
```

The `ls` command does a fine job of showing us the names of everything in Nigel's root directory,[31] but that's about all it does. The default output doesn't show us any information about the sizes or modification dates of the files or directories, or who created them. It also doesn't make it clear as to which of the listed items are files and which are directories. Another limitation is that the default output is sorted alphabetically, which might not be what we want.

This is where command-line options can help us produce all of this extra information that we might need. Command-line options in Unix are specified by using a hyphen character (-) after the command name, followed by various letters, numbers, or words. If you add the letter 'l' to the `ls` command it will give you a *l*onger output compared to the default:

```
$ ls -l /
total 36494
drwxrwxr-x+  85 root  admin       2890 Jun 28   11:35
Applications
drwxrwxr-x@  15 root  admin        510 Oct 19    2009   Developer
drwxrwxr-t+  59 root  admin       2006 Jun 23   13:04   Library
drwxr-xr-x@   2 root  wheel         68 Jun 22    2009   Network
drwxr-xr-x    4 root  wheel        136 Jun 20   13:09   System
drwxr-xr-x    6 root  admin        204 Feb 22   09:45   Users
```

[30] We think this seems like a perfectly reasonable thing to do. If you disagree, then you probably haven't spent enough time working with Unix yet.

[31] This listing displays the standard contents of a Mac computer's root directory.

```
drwxrwxrwt@   5 root  admin       170 Jun 29  11:43  Volumes
drwxr-xr-x@  39 root  wheel      1326 Jun 20  13:04  bin
drwxrwxr-t@   2 root  admin        68 Apr 15  00:26  cores
dr-xr-xr-x    3 root  wheel      4901 Jun 29  11:10  dev
lrwxr-xr-x@   1 root  wheel        11 Oct 19   2009  etc -> private/etc
dr-xr-xr-x    2 root  wheel         1 Jun 29  11:11  home
-rw-r--r--@   1 root  wheel  18661932 Jun 10  16:19  mach_kernel
dr-xr-xr-x    2 root  wheel         1 Jun 29  11:11  net
drwxr-xr-x@   6 root  wheel       204 Oct 19   2009  private
drwxr-xr-x@  64 root  wheel      2176 Jun 20  13:04  sbin
lrwxr-xr-x@   1 root  wheel        11 Oct 19   2009  tmp -> private/tmp
drwxr-xr-x@  14 root  wheel       476 Apr 21  14:38  usr
lrwxr-xr-x@   1 root  wheel        11 Oct 19   2009  var -> private/var
```

As you can see, the simple addition of `-l` makes a lot of difference to the output. For each file or directory we now see much more information and all of the output is now arranged into columns with the file or directory name in the last column. We don't need to understand all of this information at the moment, but you can hopefully see that the penultimate column includes the last modification date and time of the file or directory.

One other thing about this long-form output that we'll mention now is that the very first character describes something called the *entry type*. If it starts with a dash (-), the item is a regular file – as we can see above, there is only one file at the root level of Nigel's computer. If it starts with a 'd', then the item is a directory; if it starts with an 'l', then it is something called a *symbolic link*.[32] There are some other possibilities as well, but these are the most common ones.

More command-line options for the `ls` command

The `ls` command has many different command-line options. Here are a few examples; we encourage you to run each of these commands in one or more directories on your computer. The text following each command (after the #) is just an explanatory comment, don't type it:

```
ls -t   # sort output based on file modification date
ls -S   # sort output by size
ls -r   # reverse-sort the output
ls -R   # recursively list output of all directories below current level
ls -1   # force output to be one entry per line
```

These examples illustrate that command-line options can use lower- or upper-case letters and also use numbers (such as in the last option).

[32] Symbolic links are similar to 'shortcuts' in the Windows OS and 'aliases' in the Mac OS. Basically, a symbolic link is a file that points to another file or directory (or even another symbolic link). Any action you perform against a symbolic link will produce the same result as if you performed the same action against the original item. The exception to this is that if you delete a symbolic link, you don't delete the original item. The long listing of the `ls` command will also show you what item the symbolic link is pointing to.

You can combine multiple options together (where appropriate) – e.g., maybe you want to list items in a directory in a long format, and then reverse-sort items based on their modification date. This would mean that the most recently modified items appear at the bottom of the list. We could do this as shown here:

```
$ ls -l -t -r /
total 36494
drwxr-xr-x@    2   root   wheel        68   Jun 22   2009   Network
lrwxr-xr-x@    1   root   wheel        11   Oct 19   2009   var -> private/var
lrwxr-xr-x@    1   root   wheel        11   Oct 19   2009   tmp -> private/tmp
lrwxr-xr-x@    1   root   wheel        11   Oct 19   2009   etc -> private/etc
drwxr-xr-x@    6   root   wheel       204   Oct 19   2009   private
drwxrwxr-x@   15   root   admin       510   Oct 19   2009   Developer
drwxr-xr-x     6   root   admin       204   Feb 22   09:45  Users
drwxrwxr-t@    2   root   admin        68   Apr 15   00:26  cores
drwxr-xr-x@   14   root   wheel       476   Apr 21   14:38  usr
-rw-r--r--@    1   root   wheel  18661932   Jun 10   16:19  mach_kernel
drwxr-xr-x@   39   root   wheel      1326   Jun 20   13:04  bin
drwxr-xr-x@   64   root   wheel      2176   Jun 20   13:04  sbin
drwxr-xr-x     4   root   wheel       136   Jun 20   13:09  System
drwxrwxr-t+   59   root   admin      2006   Jun 23   13:04  Library
dr-xr-xr-x     3   root   wheel      4901   Jun 29   11:10  dev
dr-xr-xr-x     2   root   wheel         1   Jun 29   11:11  net
dr-xr-xr-x     2   root   wheel         1   Jun 29   11:11  home
drwxrwxr-x+   85   root   admin      2890   Jun 30   10:49  Applications
drwxrwxrwt@    5   root   admin       170   Jul 2    15:58  Volumes
```

In this example we specify three different command-line options for the `ls` command. Note that we could put these options in any order. In these situations, you can combine all options by using the following syntax:

```
$ ls -ltr /
```

This achieves exactly the same as the previous command, but saves you a little bit of typing. Some Unix commands have options that are mutually exclusive; you will be warned if you attempt to use the options in an incompatible manner.

One other useful command-line option for the `ls` command is the `-p` option. This option simply adjusts the default output to additionally include a forward-slash character for any items which are directories. Compare the following output to what we saw before when we first used the `ls` command in this chapter:[33]

```
$ ls -p /
Applications/  System/   cores/   mach_kernel   tmp
Developer/     Users/    dev/     net/          usr/
```

[33] Note that the 'etc,' 'tmp,' and 'var' entries do not have a trailing forward-slash because they are symbolic links. Although these can sometimes act like directories, they are considered by Unix to be a special type of file.

```
Library/        Volumes/  etc      private/      var
Network/        bin/      home/    sbin/
```

Now it becomes a lot easier to distinguish the single item which isn't a directory ('mach_kernel'). The `-p` option also works if you use the long listing (`-l` option).

Command-line options may require additional information

Let's take a very quick look at another basic Unix command. Do you want to know what the date and time is? Try the `date` command:

```
$ date
Fri Jul 2 17:17:45 PDT 2010
```

The date command has a few basic options available – e.g., we could display the current date in UTC format[34] with the `-u` option:

```
$ date -u
Sat Jul 3 00:19:39 UTC 2010
```

One of the other options for the `date` command is `-r`. Unlike the options we have seen so far, this option requires us to also specify some additional information. If we don't specify anything else, we will see something like this:

```
$ date -r
date: option requires an argument -- r
usage: date [-jnu] [-d dst] [-r seconds] [-t west]
     [-v[+|-]val[ymwdHMS]]
     [-f fmt date | [[[[[cc]yy]mm]dd]HH]MM[.ss]] [+format]
```

Buried in that cryptic-looking usage statement is a clue that the `-r` option requires some value which corresponds to 'seconds.' The `-r` option will give you a date that corresponds to the number of specified seconds that have elapsed since January 1, 1970.[35] E.g.:

```
$ date -r 123456789
Thu Nov 29 13:33:09 PST 1973
```

The point of this example is purely to illustrate that some command-line options work on their own, and others require data. You might be wondering how you can find out about what command-line options are available for any given command. Luckily, we are going to address that in the next chapter.

Problem 3.12.1 Try listing the contents of your home directory and the root directory using various command-line options for the `ls` command. Make sure you try combining different options together and try to see how the output changes.

[34] UTC refers to Coordinated Universal Time which is approximately equivalent, but not identical to Greenwich Mean Time (GMT).

[35] No, we're not quite sure why anyone would want to be able to do this, but we guess that some people must need to do it, otherwise the command-line option wouldn't exist. If you didn't already know, January 1, 1970 is the date that all Unix systems use (and many other computers too) as the starting point for what is called 'Unix time.'

Man pages 3.13

Time to man the battle stations

If Unix commands have so many options, you might be wondering how you find out what they are and what they do. Thankfully, every Unix command should have an associated 'manual' which is just a formatted page of text that describes everything about the command. These manuals are more commonly known as 'man pages.'[36]

Example 3.13.1 We can view man pages by using the Unix `man` command – e.g., if we want to see what the `whoami` command does, we just type:

```
$ man whoami
```

Explanation

What happens next is that Unix sends the contents of the manual to a Unix text-viewing program, which gives you basic controls for scrolling through the document and searching for specific text. The text-viewing program will almost certainly be a program called `less`.[37] This is what you should see if you type the above command:

```
WHOAMI(1)      BSD General Commands Manual           WHOAMI(1)
NAME
whoami -- display effective user id
SYNOPSIS
whoami
DESCRIPTION
The whoami utility has been obsoleted by the id(1) utility, and is equivalent to "id -un".
The command "id -p" is suggested for normal interactive use.

The whoami utility displays your effective user ID as a name.

EXIT STATUS
The whoami utility exits 0 on success, and >0 if an error occurs.
SEE ALSO
id(1)
BSD                    June 6, 1993                  BSD
(END)
```

Unix man pages are formatted in a standard layout which includes sections for 'name,' 'synopsis,' and 'description.' The man page for the `whoami` command is very short because this command does one thing, and one thing only … it tells you who you are.[38] If this command had any command-line options then they would all be described in the 'description' section.

[36] For readers who were hoping that man pages would help them understand men, there is no such help from Unix. Additionally, there are no woman pages of any kind.

[37] Though bear in mind that it is possible to change the program that is used as the man page viewer.

[38] This is not as stupid as it may seem. Certain Unix users (e.g., system administrators) may be in charge of multiple accounts on multiple machines. It is entirely possible to forget which user account you have logged in as.

Working with directories 3.14

Make directories, not war

Where should you create files for this book?

Over the last few chapters we've had to make some digressions in order to teach you about some important Unix concepts. Now it is time to actually start doing 'stuff,' which you will hopefully find a little more enjoyable. But before we can do anything we should first ask you to choose a place to store all of the files and directories that you will create as you work through the rest of this book. This place should ideally be a *single* directory[41] that will subsequently have more subdirectories added. For now, we suggest creating that directory in your home directory (details of how to do this will follow shortly), but you may want to save it somewhere else (external hard drive, flash drive, etc.) if you frequently work on different computers.

Naming directories

When working with a graphical file manager, we are accustomed to including spaces as part of file or folder names (e.g., 'My important text file.txt'). You can do this on a Unix system too, but it does present an additional complication. Up till now we have used the space character to separate out different parts of a Unix command – e.g., if you had to type a Unix command that requires two arguments, we would separate those arguments with one or more space characters.

So how can we have a file or directory name that also includes a space? How does Unix know that the space belongs to the directory name and isn't just separating arguments? Well, rather than tell you the solution we will instead suggest that any time you create a file or directory in Unix, you use underscore characters rather than spaces. This is a very common practice in Unix, and if we take our prior example of a text file, this would now become 'My_important_text_file.txt'.[42]

Making directories

To make a new directory we can use the appropriately named `mkdir` command.

Example 3.14.1 Let's create a 'Unix_and_Perl' directory, which we will use throughout the rest of this book. We'll create this for our fictional user 'Nigel,' and so the output you're going to see is for Nigel's computer. Your output will differ (especially if you choose a different directory name):

[41] Of course, if you prefer a little chaos in your life, you might want to instead create 36 differently named directories in 36 different locations on your hard drive.

[42] If you really need to know how to do this, you need to include a backslash character before every space character you use. This will make much more sense when you get into the Perl parts of the book later.

```
$ cd
$ pwd
/Users/nigel
$ mkdir Unix_and_Perl
$ ls -lp
drwx------+   5  nigel  staff  170   Feb  5   12:48  Desktop/
drwx------+   4  nigel  staff  136   Oct 19    2009  Documents/
drwx------+   4  nigel  staff  136   Oct 19    2009  Downloads/
drwx------+  27  nigel  staff  918   Jan 14   13:43  Library/
drwx------+   3  nigel  staff  102   Oct 19    2009  Movies/
drwx------+   3  nigel  staff  102   Oct 19    2009  Music/
drwx------+   4  nigel  staff  136   Oct 19    2009  Pictures/
drwxr-xr-x+   6  nigel  staff  204   Jan 14   13:44  Public/
drwxr-xr-x+   5  nigel  staff  170   Oct 19    2009  Sites/
drwxr-xr-x    2  nigel  staff   68   Jul  8   14:10  Unix_and_Perl/
```

Explanation

Notice that we first run the `cd` command to navigate to Nigel's home directory[43] and then run the `pwd` command to confirm its location. Then we run the `mkdir` command and simply specify the name of the directory that we want to create. Finally, we use the `-lp` options of the `ls` command to get a long listing of the directory, which also adds forward-slashes to the ends of any directory names.

Example 3.14.2 Now that we have a container directory for all of the Unix and Perl material that we might create, we can also make some more specific subdirectories:

```
$ mkdir Unix_and_Perl/Code
$ mkdir -p Unix_and_Perl/Temp/Inside_temp
$ cd Unix_and_Perl
```

Explanation

There are two things to note from this. First, the `mkdir` command can be used to create directories inside existing directories. Of course we could have also just changed directory into the 'Unix_and_Perl' directory before creating the 'Code' subdirectory. Second, you can create nested directories in one go if you use the `-p` command-line option.[44] This allows us to create a directory 'Inside_temp' plus its parent directory ('Temp') in one operation. If we had not specified the `-p` option, we would have seen an error like so:

```
mkdir: Temp: No such file or directory
```

[43] This is output from an actual Mac computer. Don't confuse this with our 'fictional' filesystem hierarchy that we showed in earlier chapters in which Nigel's home directory was located in /home.

[44] Of course, you can always find out more about this (and any other options for the `mkdir` command) by looking at its man page.

We will use the 'Code' directory for storing any Unix or Perl scripts that we write in subsequent chapters. The 'Temp' directory will be a place for trying out various Unix commands.

Removing directories

We only wanted to create the 'Inside_temp' directory to illustrate the usefulness of the `mkdir` command's `-p` option. Now we should remove it by using the `rmdir` command.

Example 3.14.3 Let's navigate into the 'Temp' directory and then remove the 'Inside_temp' directory:

```
$ cd Temp
$ ls
Inside_temp
$ rmdir Inside_temp
```

Explanation

In this case we removed the 'Inside_temp' directory while being located just one level above it. However, we could have also removed the directory without having to use the `cd` command by doing either of the following:

```
$ rmdir Temp/Inside_temp
$ rmdir /Users/nigel/Unix_and_Perl/Temp/Inside_temp
```

Hopefully you will realize that the first of these examples uses the *relative* path to the 'Inside_temp' directory, whereas the second example uses the absolute path (revisit Chapter 3.8 if you need a refresher on absolute and relative paths).

Note that the `rmdir` command will only remove *empty* directories (we'll cover how to remove directories that contain files later on). Also, you can remove a directory from anywhere *except* if you are inside the directory you want to remove.[45]

[45] Unix won't let you remove the ground from beneath your feet.

The importance of saving keystrokes 3.15

Or the art of accomplishing more by typing less

When we interact with the world of Unix, most of that interaction occurs via the keyboard. If you can reduce the amount of typing you have to do in order to accomplish a task then this is good for you in two ways. First, it makes you more productive because you are spending more time running commands and getting results, and less time typing their names. Second, and more important, it will help you minimize the amount of typing you do, which in turn will help lessen the risks of developing a repetitive strain injury (RSI).[46] In addition to having very short command names, Unix offers a few other ways to ease the load on your digits.

Command-line completion

Perhaps the most important time-saver to learn is something called *command-line completion*. This allows you to automatically complete the names of files, directories, and programs as you type them. If you type enough letters that uniquely identify the name of something and then press tab[47] … Unix will do the rest.

Example 3.15.1 Type the letters 'tou' and then press the tab key on your keyboard. In the following examples, we'll use `<tab>` as a shortcut way of saying 'press tab.'

```
$ tou<tab>
$ touch
```

Explanation

If this works, you should see the letters 'ch' become magically added to the three letters you have already typed. This forms the name of the Unix `touch` command (which we will learn more about in a later chapter). In this case, command-line completion will occur because there are no other standard Unix commands that start with the letters 'tou'. If this didn't work it might be because you have a non-standard Unix command on your system that also starts with 'tou', or that there is a file or directory in your current location that starts with 'tou'. When there are no possible completions for the letters you have already typed, you may hear a beep.[48]

Command-line completion can be used to save time when typing program names, but it is equally useful when working with files and directories. If you have yet to type the name of a Unix command, then tab-completion will attempt to complete

[46] Please do not underestimate the health risks that can result from overusing (or incorrectly using) a keyboard and mouse. Unix and most programming languages make heavy use of various punctuation symbols on your keyboard, and this usually means your fingers end up being stretched a little more than if you were just typing 'regular' text. If you routinely experience pain or discomfort while using a keyboard, you should *stop* typing and see your doctor.

[47] It is possible that in some Unix shells, other keys will be used to trigger the completion, but the tab key is the most common trigger key, which is why command-line completion is also known as *tab-completion*.

[48] Some, but not all, Unix systems will use a beep sound to indicate errors or to serve as a warning/reminder.

from a list of all known *commands and programs* on the system. However, if you have already typed the name of a valid command, then you can use command-line completion to finish the name of whatever *file or directory* is required by the command:

Example 3.15.2 We will navigate into our 'Unix_and_Perl' directory using the `cd` command, but we want to save as many keystrokes as possible when typing the directory name:

```
$ cd U<tab>
$ cd Unix_and_Perl/
```

Explanation
In this example we only have to type five characters (c + d + space + U + tab) rather than the full 16 characters that we would otherwise have had to type. If this doesn't work for you, it is most likely because you have another file or directory in your current directory that starts with the letter 'U'. Try typing one more letter at a time, and pressing tab after each successive key press.

Another way you can use command-line completion is to press tab *twice* to show a list of all possible completions. The contents of such a list will depend on how many characters of a file, directory, or program name have already been typed. It will also depend on whether you have yet typed a full Unix command. You can even press tab twice before typing anything at all. If you do this you will probably see a warning message like this:

```
Display all 1712 possibilities? (y or n)
```

This suggests that, on our computer, there are 1712 different Unix commands and programs that can be run.

Example 3.15.3 Now we will use tab-completion to browse through some directories that are at the root level of the computer:

```
$ ls<tab><tab>
ls  lsbom   lsdistcc      lsm     lsof    lsvfs
$ ls /u<tab>
$ ls /usr/b<tab>
$ ls /usr/bin/auto<tab><tab>
autoconf       autom4te     automake-1.10  autoreconf
autoupdate
autoheader     automake     automator      autoscan
```

Explanation
The first part of this example shows that even when you have typed the name of an existing command (in this case `ls`), there may be other Unix commands that match the

Ctrl + a	move to start of line
Ctrl + e	move to end of line
Ctrl + w	delete previous word on command line
Ctrl + l	clear screen[50]

Problem 3.15.1 Practice navigating through some directories using the `cd` command, but make sure you use tab-completion to type every directory name.

Problem 3.15.2 Practice accessing your command history (using either the history command, or up-arrow navigation). Make sure you can repeat an older command.

[50] Note that there is also a simple Unix command, `clear`, which does the same thing.

Moving and renaming files 3.16

Or how you can learn to move heaven and Earth

The next few chapters, including this one, will deal with Unix commands that help to work with files, i.e., commands that will allow us to move, copy, rename, delete, and view files. In order to learn how to use these commands, we will need to have some files to play with. The Unix command `touch` lets us easily create some empty files that we can work with.[51]

Example 3.16.1 Let's create two new files in your newly created 'Unix_and_Perl' directory. Remember to always use tab-completion when typing file and directory names:

```
$ cd ~/Unix_and_Perl
$ ls
Code    Temp
$ touch heaven.txt
$ touch earth.txt
$ ls -l
total 0
drwxr-xr-x 2   nigel   staff  68   Jul   8    14:55   Code
drwxr-xr-x 2   nigel   staff  68   Jul   8    16:19   Temp
-rw-r--r-- 1   nigel   staff   0   Jul  19    15:10   earth.txt
-rw-r--r-- 1   nigel   staff   0   Jul  19    15:10   heaven.txt
```

Explanation

We first ensure that we are in the 'Unix_and_Perl' directory, which in this case is one level below the home directory. Knowing this allows us to navigate there by using the `~/Unix_and_Perl` syntax (see Chapter 3.9 for more information on using the tilde character [~] to refer to your home directory).

We then use the `touch` command to create two files which we give a .txt extension.[52] The `ls -l` command shows a long listing which confirms that the two new files exist and that they are zero bytes in size.[53] If we had wanted to, we could have also created both files in one step:

```
$ touch heaven.txt earth.txt
```

The `mv` command

If you want to move a file in Unix, you need to use the `mv` command. Whenever we use commands like `mv`, we must always bear in mind the two concepts of *source* and *target*. The source is the location of the file (or directory) that we want to move; the target is the

[51] This command does some other things as well as creating blank files. Look at its man page if you want to know more.

[52] This is not strictly necessary at this stage, because the files do not contain any text. Indeed, they do not contain anything at all.

[53] The size of files (in bytes) is listed in the fifth column of the long output. Every character in a file increases its size by one byte.

location of the place where we want to move the file. Commands such as mv will always expect you to specify a source and target location (in that order).

Example 3.16.2 Let's move these two new files into the 'Temp' directory that we created previously (Chapter 3.14):

```
$ ls
Code             Temp         earth.txt      heaven.txt
$ mv earth.txt   Temp/
$ mv heaven.txt  Temp/

$ ls
Code        Temp

$ ls Temp/
earth.txt          heaven.txt
```

Explanation

We use the mv command twice, once for each file that we want to move. The *target* location for the move is the 'Temp' directory. After moving the files, we confirm that the current directory no longer has the files and that they are instead in the 'Temp' directory. If you use tab-completion to type the name 'Temp' Unix will automatically add the trailing forward-slash character. This helps to remind you that 'Temp' is a directory and not a file.[54]

Renaming files

In the last example, the destination for the mv command was a directory name ('Temp'). However, the target could have also been a (different) file name, rather than a directory. This is how you can use mv to rename files.

Example 3.16.3 Let's make a new file and move it while renaming it at the same time:

```
$ touch rags
$ ls
Code       Temp     rags
$ mv rags  Temp/riches
$ ls
Code     Temp
$ ls Temp/
earth.txt       heaven.txt    riches
```

[54] When working with directories, you do not need to add a trailing forward-slash character to the name of the last directory in any path, but it doesn't hurt. This is another reason why using tab-completion is such a good habit to get into, because it will always add the slash character for you. See the next chapter for more details.